eBook Gratuit

APPRENEZ scheme

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#scheme

Table des matières

À propos

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.

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

Chapitre 1: Commencer avec le schéma

Remarques

Cette section fournit une vue d'ensemble du schéma et de la raison pour laquelle un développeur peut vouloir l'utiliser.

Il devrait également mentionner tous les grands sujets dans le schéma, et établir un lien avec les sujets connexes. La documentation pour le schéma étant nouvelle, vous devrez peut-être créer des versions initiales de ces rubriques connexes.

Examples

Installer le Scheme de POULET

CHICKEN est un interpréteur Scheme et un compilateur avec son propre système de module d'extension appelé "œufs". Il est capable de compiler Scheme en code natif en compilant d'abord Scheme en C.

Installation

Debian ou Ubuntu ou d'autres distributions dérivées:

```
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 avec Homebrew:

```
brew install chicken
```

OpenBSD

```
doas pkg_add -vi chicken
```

Microsoft Windows

- Installez MSYS2
- Exécuter le shell MSYS2 MinGW-w64
- Installer des prérequis en exécutant:

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

- Téléchargez la dernière version de l'archive en tapant:

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

- Extraire l'archive en exécutant `tar xvf chicken.tar.gz`
- Entrez le répertoire extrait, par exemple en tapant `cd chicken-4.11.0`
- Exécuter `make PLATFORM=mingw-msys install`

Si vous avez des problèmes pour exécuter `csi` , essayez plutôt d'exécuter `winpty csi`

Utiliser POULET

Pour utiliser le schéma REPL de CHICKEN, tapez `csi` sur la ligne de commande.

Pour compiler un programme Scheme en utilisant CHICKEN, exécutez `csc program.scm` , qui créera un `program` nommé exécutable dans le répertoire en cours.

Installation de modules

Chicken Scheme a beaucoup de modules qui peuvent être parcourus dans l'index des œufs . Les œufs sont des modules de schéma qui seront téléchargés puis compilés par chicken-scheme. Dans certains cas, il peut être nécessaire d'installer des dépendances externes à l'aide de votre gestionnaire de paquets habituel.

Vous installez les oeufs choisis avec cette commande:

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

Utiliser le REPL

Vous souhaiterez peut-être ajouter le support `readline` à votre REPL pour que l'édition des lignes dans `csi` se comporte mieux que prévu.

Pour ce faire, exécutez `sudo chicken-install readline` , puis créez un fichier nommé `~/.csirc` avec le contenu suivant:

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

Installation de mit-scheme

Voici des exemples d'installation de MIT / GNU Scheme :

Installation Debian / Ubuntu:

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

Installation manuelle:

Téléchargez le binaire Unix directement depuis le projet GNU , puis suivez les instructions de la page Web officielle :

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

# move into the directory
cd mit-scheme/src

# configure the software
./configure
```

Par défaut, le logiciel sera installé dans `/usr/local` , dans les sous-répertoires bin et lib. Si vous voulez qu'il soit installé ailleurs, par exemple `/opt/mit-scheme` , passez l'option `--prefix` au script configure, comme dans `./configure --prefix=/opt/mit-scheme` .

Le script configure accepte tous les arguments normaux pour de tels scripts et en accepte d'autres spécifiques à MIT / GNU Scheme. Pour voir tous les arguments possibles et leur signification, exécutez la commande `./configure --help` .

```
# build
make compile-microcode

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

Windows 7 :

L' exécutable auto-installable se trouve sur le site officiel .

> MIT / GNU Scheme est distribué en tant qu'exécutable auto-installable. L'installation du logiciel est simple. Exécutez simplement le fichier téléchargé et répondez aux questions de l'installateur. Le programme d'installation vous permettra de choisir le répertoire dans lequel MIT / GNU Scheme doit être installé, ainsi que le nom du dossier dans lequel les raccourcis doivent être placés.

Lire Commencer avec le schéma en ligne: https://riptutorial.com/fr/scheme/topic/851/commencer-avec-le-schema

Chapitre 2: Implémentation de différents algorithmes de tri

Examples

Tri rapide

Quicksort est un algorithme de tri commun avec une complexité de cas moyenne de `O(n log n)` et une complexité de cas le plus défavorable de `O(n^2)` . Son avantage par rapport aux autres méthodes `O(n log n)` est qu'il peut être exécuté sur place.

Quicksort divise l'entrée sur une valeur de pivot choisie, en séparant la liste des valeurs inférieures et supérieures (ou égales) au pivot. La division de la liste se fait facilement avec le `filter` .

En utilisant cela, une implémentation Scheme de Quicksort peut ressembler à ceci:

```
(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
```

Tri par fusion

Merge Sort est un algorithme de tri commun avec une complexité de cas moyenne de `O(n log n)` et une complexité de cas le plus défavorable de `O(n log n)` . Bien qu'il ne puisse pas être exécuté sur place, il garantit la complexité de `O(n log n)` dans tous les cas.

Fusionner le tri divise de manière répétée l'entrée en deux, jusqu'à ce qu'une liste vide ou une liste d'éléments uniques soit atteinte. Après avoir atteint le bas de l'arbre de fractionnement, celui-ci se remet à remonter, fusionnant les deux divisions triées les unes avec les autres, jusqu'à ce qu'une seule liste triée soit laissée.

En utilisant ceci, une implémentation Scheme de Merge Sort peut ressembler à ceci:

```
;; 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)))))))
```

Lire Implémentation de différents algorithmes de tri en ligne:
https://riptutorial.com/fr/scheme/topic/3191/implementation-de-differents-algorithmes-de-tri

Chapitre 3: Paires

Introduction

Une paire est l'un des types de données les plus élémentaires du schéma. On appelle aussi généralement les cellules contre.

Examples

Créer une paire

Une paire peut être créée avec la fonction `cons` . Le nom de la fonction signifie *constructeur* . Dans Scheme, tout est basé sur des paires.

```
(cons a b)
```

La fonction retourne une paire contenant l'élément `a` et `b` . Le premier paramètre de `cons` est appelé `car` (Content Address Register) et le second argument est le `cdr` (Content Decrement Register).

Accédez à la voiture de la paire.

Les données de la paire sont accessibles avec des fonctions utilitaires. Pour accéder à la `car` , nous devons utiliser la fonction `car` .

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

Nous pouvons également vérifier l'égalité suivante:

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

Accédez au cdr de la paire

Pour accéder au cdr, il faut utiliser la fonction cdr.

(cdr (contre ab))

    b

Nous pouvons également vérifier l'égalité suivante:

(éq? b (cdr (contre ab)))

    #t

Créer une liste avec des paires

List in Scheme n'est rien d'autre qu'une série de paires imbriquées les unes dans les autres dans le `cdr` d'un `cons` . Et le dernier `cdr` d'une liste correcte est la liste vide `'()` .

Pour créer la liste `(1 2 3 4)` , nous aurions quelque chose comme ceci:

```
(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)
```

Comme vous pouvez le voir, une liste dans le schéma est une liste chaînée faite de paires. Pour cette raison, l'ajout d'un objet au début de la liste ne prend presque pas de temps, mais l'ajout d'un élément à la fin de la liste force l'interpréteur à parcourir toute la liste.

Lire Paires en ligne: https://riptutorial.com/fr/scheme/topic/8190/paires

Chapitre 4: Schéma Macros

Examples

Macros hygiéniques et référentiellement transparentes avec règles de syntaxe

Le plus grand avantage de LISP et de Scheme par rapport aux autres langages de programmation classiques est leur macro-système. Contrairement au préprocesseur C et aux autres langages macro, les macros Scheme prennent le code analysé en entrée et renvoient le code développé en sortie. C'est l'une des applications de l'expression «code is data» de Scheme et c'est ce qui rend le langage si puissant.

Les macros dans Scheme sont créées avec `define-syntax` , qui peut définir une macro de différentes manières. La méthode la plus simple consiste à utiliser `syntax-rules` , qui utilisent la correspondance de modèle pour transformer le code d'entrée en code de sortie.

Cet exemple crée un `for item in list` simple `for item in list` et `for list as item` syntaxe d' `for list as item` permettant de survoler les éléments d'une liste:

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

Ces deux macros peuvent alors être utilisées comme suit, offrant un style plus impératif:

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

L'exécution du code fournira le résultat attendu:

```
Hello Alice
Hello Bob
```

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

L'erreur la plus courante à rechercher est de ne pas transmettre les valeurs correctes à une macro, ce qui entraîne souvent un message d'erreur inutile qui s'applique au formulaire développé au lieu de l'appel de macro.

Les `for` les définitions de syntaxe ci - dessus ne vérifient pas s'ils sont passés d' un identifiant et une liste, donc passer tout autre type se traduira par une erreur de pointage à la `for-each` appel au lieu du `for` appel. Déboguer cela défait le but de la macro, il est donc à l'utilisateur de mettre les contrôles et de signaler les erreurs d'utilisation, qui peuvent ensuite être interceptées au moment de la compilation.

Lire Schéma Macros en ligne: https://riptutorial.com/fr/scheme/topic/3024/schema-macros

Chapitre 5: Sortie d'entrée dans le schéma

Introduction

Les entrées et sorties dans le schéma sont généralement gérées par les ports. Un port est une structure de données utilisée pour interagir avec le monde extérieur au schéma. Un port n'est pas limité aux fichiers mais peut être utilisé pour lire / écrire sur les sockets. À certains égards, l'objet port est une sorte d'objet universel qui peut non seulement manipuler des fichiers et des sockets, mais aussi toute opération de lecture / écriture avec le système d'exploitation. Par exemple, on pourrait implémenter un port capable d'écrire sur une imprimante ou même de contrôler une machine CNC à partir d'un schéma utilisant un port.

Examples

Créer un port d'entrée

Un port d'entrée peut être créé de plusieurs manières, mais la méthode commence généralement par `open-input-` .

Port de chaîne

Vous pouvez utiliser une chaîne en tant que port en utilisant `open-input-string` . Cela créera un port qui pourra lire à partir de la chaîne.

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

Port de fichier

Vous pouvez ouvrir un fichier pour le lire avec `open-input-file` .

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

Lecture depuis un port d'entrée

La lecture depuis un port d'entrée peut se faire de plusieurs manières. Nous pouvons utiliser la méthode de `read` utilisée par le REPL. Il va lire et interpréter les expressions séparées par des espaces.

Prenons l'exemple du port de chaîne ci-dessus. Nous pouvons lire depuis le port comme ceci:

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

Nous pouvons lire un port en tant que `char` utilisant la méthode spéciale `read-char` . Cela renverra un seul caractère du port que nous lisons.

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

Lire Sortie d'entrée dans le schéma en ligne: https://riptutorial.com/fr/scheme/topic/8188/sortie-d-entree-dans-le-schema

Chapitre 6: Syntaxe

Examples

S-Expression

Une expression dans Scheme est ce qui va être exécuté. Une expression S, comme on l'appelle habituellement, commence par un `(` et se termine par un `)` . Le premier membre de l'expression est ce qui va être exécuté. Les membres suivants de l'expression sont les paramètres qui seront envoyés à l'expression lors de l'évaluation de l'expression.

Par exemple, ajouter des nombres:

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

Dans ce cas, `+` est un symbole pour une fonction d' *ajout* qui prend plusieurs paramètres. `1` , `2` et `3` sont envoyés à la fonction `+` .

S-Expression peut contenir des expressions S comme paramètres, comme illustré dans l'exemple suivant:

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

Ce qui peut être lu comme si `x` est inférieur à `y` retourne `x` sinon retourne `y` . Dans cet exemple, nous évaluons l'expression de la condition, en fonction de la valeur résolue, x ou y seront renvoyés. Il pourrait être évalué à cette

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

Un exemple moins évident pour les débutants est d'avoir une expression S dans le premier membre d'une expression S. De cette façon, nous pouvons changer le comportement d'une méthode en modifiant la fonction qui sera appelée sans avoir à créer de branches avec les mêmes paramètres. Voici un exemple rapide d'une expression qui ajoute ou soustrait des nombres si x est inférieur à y.

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

Si `x` est inférieur à `y` , l'expression sera évaluée comme suit:

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

autrement

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

Comme vous pouvez le voir, Scheme permet au programmeur de construire un morceau de code complexe tout en donnant au programmeur les outils nécessaires pour éviter la duplication de code. Dans d'autres langues, nous pourrions voir le même exemple écrit en tant que tel:

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

Le problème avec cette méthode est que nous dupliquons beaucoup de code alors que la seule chose qui change est la méthode appelée. Cet exemple est assez simple mais avec plus de conditions, nous pourrions voir beaucoup de lignes similaires dupliquées.

Macro simple laisser

Les expressions let dans le schéma sont en fait des macros. Ils peuvent être exprimés avec des lambda. Un simple let pourrait ressembler à ceci:

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

Il retournera 3 car la valeur de la dernière expression du corps let est renvoyée. Comme vous pouvez le voir, une expression-let est en train d'exécuter quelque chose. Si nous traduisons cette partie de code avec lambdas, nous aurons quelque chose comme ceci:

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

Ici, nous pouvons voir que nous appelons le lambda anonyme avec 1 et 2 directement. Donc, le résultat dans ce cas est également 3.

Dans cette optique, nous comprenons qu'une expression let est composée de deux parties. Il a des paramètres et un corps comme un lambda, mais la différence est que let expression est appelée juste après son évaluation.

Pour expliquer comment une expression laisser fonctionner d'un résumé à une vue concrète, cela ressemblerait à ceci.

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

Les paramètres sont une liste de paires de `(name value)` à utiliser dans le corps du `let` .

Pourquoi utiliser let expression?

Les expressions sont particulièrement utiles pour stocker des variables dans une méthode, tout comme les initialisations de variables dans les langages similaires à c. Il est favorable à

l'utilisation de `define` parce que, à partir de l'expression let, les variables ont disparu ... L'utilisation d'une définition ajoute en fait une variable à l'environnement d'exécution actuel. Les variables ajoutées à l'environnement global ne peuvent pas être supprimées. Soit expression peut être utilisé n'importe où. Il peut également être utilisé pour créer des variables fantômes sans toucher aux portées parents.

Par exemple:

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

Il va imprimer:

```
2
1
1
```

Dans ce cas, `x` est défini avec 1, puis fantôme par le `x` dans le second `let` avec la valeur `2` . La variable `y` est initiée avec la valeur `x` de la portée parent. Une fois `let` expression interne `let` est exécutée, elle affiche la valeur initiale de `x` avec 1. L'expression `let` interne n'a pas modifié la valeur de l'étendue parent.

Chaque fois que vous avez besoin d'initialiser des variables, vous devez utiliser les expressions let comme ceci:

```
(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))
```

Ici, dans cet exemple, les variables sont initialisées et utilisées plusieurs fois dans le bloc de code. Et lorsque l'expression let est terminée, les variables sont automatiquement libérées car elles ne sont plus nécessaires.

Syntaxe pointillée pour les paires

Il y a une syntaxe particulière qui nous permettent d'écrire `cons` la cellule d'une manière plus compacte que d' utiliser le `cons` constructeur.

Une paire peut être écrite comme telle:

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

La grande différence est que nous pouvons créer des `pairs` aide de devis. Sinon, Scheme créerait une liste appropriée `(1 . (2 . '()))` .

La syntaxe à points oblige l'expression à n'avoir que 2 membres. Chaque membre peut être de tout type, y compris des paires.

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

Notez que la liste incorrecte doit être affichée avec un point à la fin pour indiquer que le `cdr` de la dernière paire de la liste n'est pas la liste vide `'()` .

Cette façon de montrer des listes est parfois déroutante car l'expression suivante serait exprimée comme on ne s'y attendrait pas.

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

Depuis la liste habituellement sauter le `.` , le premier argument de la liste serait `(1 . 2)` , le deuxième argument serait `3` mais puisque la liste est incorrecte, la dernière `.` est montré pour montrer que le dernier élément de la liste n'est pas `'()` . Même si les données sont présentées différemment, les données internes sont telles qu'elles ont été créées.

Lire Syntaxe en ligne: https://riptutorial.com/fr/scheme/topic/5989/syntaxe

Crédits

S. No	Chapitres	Contributeurs
1	Commencer avec le schéma	Candy Gumdrop, Community, eyqs, Loïc Faure-Lacroix, Reut Sharabani, Will Ness
2	Implémentation de différents algorithmes de tri	Billy Brown
3	Paires	Loïc Faure-Lacroix
4	Schéma Macros	Billy Brown
5	Sortie d'entrée dans le schéma	Loïc Faure-Lacroix
6	Syntaxe	Loïc Faure-Lacroix