FREE eBook

LEARNING scheme

Free unaffiliated eBook created from
Stack Overflow contributors.

#scheme

Table of Contents

About

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

Chapter 1: Getting started with scheme

Remarks

This section provides an overview of what scheme is, and why a developer might want to use it.

It should also mention any large subjects within scheme, and link out to the related topics. Since the Documentation for scheme is new, you may need to create initial versions of those related topics.

Examples

Installing CHICKEN Scheme

CHICKEN is a Scheme interpreter and compiler with its own extension module system called "eggs". It is capable of compiling Scheme to native code by first compiling Scheme to C.

Installing

Debian or Ubuntu or other derived distros:

```
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 with Homebrew:

```
brew install chicken
```

OpenBSD

```
doas pkg_add -vi chicken
```

Microsoft Windows

- Install MSYS2
- Run the MSYS2 MinGW-w64 Shell
- Install some prerequesites by running:

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

- Download the latest release tarball by typing:

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

- Extract the tarball by running `tar xvf chicken.tar.gz`
- Enter the extracted directory, for example by typing `cd chicken-4.11.0`
- Run `make PLATFORM=mingw-msys install`

If you have trouble running `csi`, try instead running `winpty csi`

Using CHICKEN

To use the CHICKEN Scheme REPL, type `csi` at the command line.

To compile a Scheme program using CHICKEN, run `csc program.scm`, which will create an executable named `program` in the current directory.

Installing modules

Chicken Scheme has a lot of modules that can be browsed in the egg index. Eggs are scheme modules that will be downloaded and then compiled by chicken-scheme. In some cases, it might be necessary to install external dependencies using your usual package manager.

You install the chosen eggs with this command:

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

Making use of the REPL

You may wish to add `readline` support to your REPL to make line editing in `csi` behave more like you might expect.

To do this, run `sudo chicken-install readline`, and then create a file named `~/.csirc` with the following contents:

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

Installing mit-scheme

The following are examples of how to install MIT/GNU Scheme:

Debian/Ubuntu installation:

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

Manual installation:

Download the Unix binary directly from the GNU Project, then follow the instructions from the official webpage:

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

# move into the directory
cd mit-scheme/src

# configure the software
./configure
```

By default, the software will be installed in `/usr/local`, in the subdirectories bin and lib. If you want it installed somewhere else, for example `/opt/mit-scheme`, pass the `--prefix` option to the configure script, as in `./configure --prefix=/opt/mit-scheme`.

The configure script accepts all the normal arguments for such scripts, and additionally accepts some that are specific to MIT/GNU Scheme. To see all the possible arguments and their meanings, run the command `./configure --help`.

```
# build
make compile-microcode

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

Windows 7:

The self-installing executable can be found in the official website.

> MIT/GNU Scheme is distributed as a self-installing executable. Installation of the

software is straightforward. Simply execute the downloaded file and answer the installer's questions. The installer will allow you to choose the directory in which MIT/GNU Scheme is to be installed, and the name of the folder in which the shortcuts are to be placed.

Read Getting started with scheme online: https://riptutorial.com/scheme/topic/851/getting-started-with-scheme

Chapter 2: Implementation of different sortings algorithms

Examples

Quicksort

Quicksort is a common sorting algorithm with an average case complexity of `O(n log n)` and a worst case complexity of `O(n^2)`. Its advantage over other `O(n log n)` methods is that it can be executed in-place.

Quicksort splits the input on a chosen pivot value, separating the list into those values that are less than and those values that are greater than (or equal to) the pivot. Splitting the list is easily done with `filter`.

Using this, a Scheme implementation of Quicksort may look like the following:

```
(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
```

Merge Sort

Merge Sort is a common sorting algorithm with an average case complexity of `O(n log n)` and a worst case complexity of `O(n log n)`. Although it cannot be executed in-place, it guarantees `O(n log n)` complexity in all cases.

Merge Sort repeatedly splits the input in two, until an empty list or single-element list is reached. Having reached the bottom of the splitting tree, it then works its way back up, merging the two sorted splits into each other, until a single sorted list is left.

Using this, a Scheme implementation of Merge Sort may look like the following:

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

Read Implementation of different sortings algorithms online:
https://riptutorial.com/scheme/topic/3191/implementation-of-different-sortings-algorithms

Chapter 3: Input Output in Scheme

Introduction

Input and Output in scheme is usually handled trough ports. A port is a data structure which is used to interact with the world outside Scheme. A Port isn't limited to files but can be used to read/write to sockets. In some ways, the port object is some kind of universal object that can not only manipulate file and sockets but any kind of read/write operation with the OS. For example, one could implement a port that can write to a printer or even control a CNC machine from Scheme using a port.

Examples

Create an input port

An input port can be created in many ways, but usually the method starts with `open-input-`.

String port

You can use a string as a port using `open-input-string`. It will create a port that will be able to read from the string.

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

File port

You can open a file for reading with `open-input-file`.

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

Read from an input port

Reading from an input port can be done in many ways. We can use the `read` method used by the REPL. It will read and interpret space separated expressions.

Taking the example from the string port above. We can read from the port like this:

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

We can read from a port as `char` using the special method `read-char`. This will return a single char from the port that we're reading from.

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

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

Chapter 4: Pairs

Introduction

A Pair is one of the most basic data type in scheme. It is also usually called cons cells.

Examples

Create a pair

A pair can be create with the `cons` function. The name of the function stand for *constructor*. In Scheme, everything is pretty much based on pairs.

```
(cons a b)
```

The function return a pair containing the element `a` and `b`. The first parameter of `cons` is called `car` (Content Address Register) and the second argument is the `cdr` (Content Decrement Register).

Access the car of the pair.

The data in the pair can be accessed with utility functions. To access the `car`, we have to use the `car` function.

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

Also we can verify the following equality:

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

Access the cdr of the pair

To access the cdr, we have to use the cdr function.

(cdr (cons a b))

    b

Also we can verify the following equality:

(eq? b (cdr (cons a b)))

    #t

Create a list with pairs

List in scheme are nothing else than a series of pairs nested in each other in the `cdr` of a `cons`. And the last `cdr` of a proper list is the empty list `'()`.

To create the list `(1 2 3 4)`, we'd have something like this:

```
(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)
```

As you can see, a list in scheme is a linked list made out of pairs. For that reason, adding an object to the front of the list takes almost no time, but appending an element at the end of the list forces the interpreter to walk accross the whole list.

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

Chapter 5: Scheme Macros

Examples

Hygienic and referentially-transparent macros with syntax-rules

LISP and Scheme's greatest advantage over other mainstream programming language is their macro system. Unlike the C preprocessor and other macro languages, Scheme macros take parsed code as input and return expanded code as output. This is one of the applications of Scheme's “code is data” phrase, and it is what makes the language so powerful.

Macros in Scheme are created with `define-syntax`, which can define a macro in a number of ways. The simplest method is to use `syntax-rules`, which uses pattern-matching to transform the input code into the output code.

This example creates a simple `for item in list` and `for list as item` syntax for looping over elements in a list:

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

These two macros can then be used as follows, providing a more imperative style:

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

Running the code will provide the expected output:

```
Hello Alice
Hello Bob
Hello Eve
```

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

The most common mistake to look out for is not passing the correct values to a macro, which will often result in an unhelpful error message that applies to the expanded form instead of the macro call.

The `for` syntax definitions above do not check whether they are passed an identifier and a list, so passing any other type will result in an error pointing to the `for-each` call instead of the `for` call. Debugging this defeats the purpose of the macro, so it is up to the user to put the checks in there and report usage errors, which can then be caught at compile time.

Read Scheme Macros online: https://riptutorial.com/scheme/topic/3024/scheme-macros

Chapter 6: Syntax

Examples

S-Expression

An expression in Scheme is what is going to get executed. A S-expression, as it's usually called starts with a `(` and end with a `)`. The first member of the expression is what is going to get executed. The following member of the expression are the parameters that will be sent to the expression during the evaluation of the expression.

For example adding numbers:

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

In this case, `+` is a symbol to a *add* function that takes multiple parameters. `1`, `2` and `3` are sent to the `+` function.

S-Expression may contain S-Expressions as parameters as shown in the following example:

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

Which can be read as if `x` is less than `y` return `x` else return `y`. In this example we evaluate the condition expression, depending on the resolved value, either x or y will be returned. It could be evaluated to this

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

A less obvious example for beginners is to have a S-Expression as part of the first member of a S-Expression. This way, we can change the behaviour of a method by changing the function that will be called without having to create branches with the same parameters. Here's a quick example of an expression that either add or substract numbers if x is below y.

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

If `x` is below `y`, the expression will be evaluated as:

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

otherwise

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

As you can see, Scheme allow the programmer to build up complex piece of code while giving the programmer the tools to prevent duplicating code. In other languages we could see the same example written as such:

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

The problem with this method is that we duplicate a lot of code while the only thing that change is the method being called. This example is fairly simple but with more condition we could see a lot of similar lines duplicated.

Simple let macro

The let expressions in scheme are in fact macros. They can be expressed with lambdas. A simple let might look like this:

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

It will return 3 as the value of the last expression of the let body is returned. As you can see, a let-expression is actually executing something. If we translate this part of code with lambdas, we'd get something like this:

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

Here we can see that we're calling the anonymous lambda with 1 and 2 directly. So the result in this case is also 3.

With that in mind, we understand that a let expression is composed of 2 parts. It has parameters and a body like a lambda has, but the difference is that let expression are called after right after their evaluation.

To explain how a let expression work from an abstract to concrete view, it would look like this.

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

The parameters are a list of pair of `(name value)` to be used in the body of the `let`.

Why use let expression?

Let expressions are particularly useful to store variables in a method just like initializations of variable in c like languages. It is favorable to the use of `define` because out of the let expression, the variables are gone... Using a define is actually adding a variable to the current execution environment. Variables that are added to the global environment cannot be removed. Let expression are safe to use anywhere. It can also be used to ghost variables without touching the

parent scopes.

For example:

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

It will print:

```
2
1
1
```

In this case, `x` is defined with 1, then ghosted by the `x` in the second `let` with the value `2`. The variable `y` is initiated with the value `x` of the parent scope. After the inner `let` expression is executed, it display the initial value of `x` with 1. The inner `let` expression didn't change the value of the parent scope.

Whenever you need to initialize variables, you should be using let expressions like this:

```
(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))
```

Here in this example, the variables are initialized and used multiple time in the code block. And when the let expression is finished, the variables are automatically freed as they are not necessary anymore.

Dotted syntax for pairs

There is a particular syntax that allow us to write `cons` cell in a more compact way than using the `cons` constructor.

A pair can be written as such:

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

The big difference is that we can create `pairs` using quote. Otherwise, Scheme would create a proper list `(1 . (2 . '()))`.

The dot syntax force the expression to have only 2 members. Each member can be of any type including pairs.

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

Note that the improper list should be displayed with a dot at the end to show that the `cdr` of the last pair of the list isn't the empty list `'()`.

This way of showing lists is sometime confusing as the following expression would be expressed not like one would expect it.

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

Since list usually skip the `.`, the first argument of the list would be `(1 . 2)`, the second argument would be `3` but since the list is improper, the last `.` is shown to show that the last element of the list isn't `'()`. Even thought, the data is shown in a different way, the internal data is as it was created.

Read Syntax online: https://riptutorial.com/scheme/topic/5989/syntax

Credits

S. No	Chapters	Contributors
1	Getting started with scheme	Candy Gumdrop, Community, eyqs, Loïc Faure-Lacroix, Reut Sharabani, Will Ness
2	Implementation of different sortings algorithms	Billy Brown
3	Input Output in Scheme	Loïc Faure-Lacroix
4	Pairs	Loïc Faure-Lacroix
5	Scheme Macros	Billy Brown
6	Syntax	Loïc Faure-Lacroix