The Little Schemer Simplified

Deepak Venkatesh

October 17, 2025

Contents

Note:

These are my personal notes created to deepen my understanding of Lisp and programming in general. 'The Little Schemer' (4th edition) by Daniel Friedman and Matthias Felleisen is a remarkable book that teaches programming concepts in a unique and playful way. It builds from first principles using only a small set of primitives, showing how powerful ideas — such as recursion, functional programming, lambda functions, and interpreters — can be expressed using just those few building blocks. I am making this public since other beginners could benefit from these notes.
While the book uses Scheme, I initially worked it out using Common Lisp and had adapted the examples accordingly. Now I am working through it using Racket, a modern descendent of Scheme. Despite its lighthearted tone, the book is far from an easy read — it demands close attention and careful thought. My advice to anyone who wants to learn lisp is to first work through Professor David Touretzky's book titled 'Common Lisp A Gentle Introduction to Symbolic Computing' (2nd ed.). It's a great book for introduction to programming not just Lisp.
All mistakes in these notes, whether typographical or conceptual, are entirely my own. Any misinterpretations are as well. I am still new to both Lisp and programming.

Hardware and Software used for this study

- Language: Racket (a Scheme)
- Editor: Dr Racket
- emacs for org notes
- Macbook Pro 2019 2.6 Ghz I7 Intel chip with 16 GB RAM

1 Foreword

By Gerald Sussman of MIT, co-author of the book SICP (the wizard book).

Key Takeaway: *In order to be creative one must first gain control of the medium.*

- Core skills are the first set of things required to master any pursuit.
- Deep understanding is required to visualize beforehand the program which will be written.
- Lisp provides freedom and flexibility (this is something which will only come in due course of time, as we keep learning more about programming).
- Lisp was initially conceived as a theoretical vehicle for recursion and symbolic algebra (this is the algebra we have been taught in school such as $(a + b)^2 = a^2 + b^2 + 2ab$).
- In Lisp procedures are first class. Procedures are essentially a 'variant' of functions. A mathematical function maps a given input to an output (domain - range/co-domain) but a procedure is a process to arrive at the result via computation.
- First Class basically means that the procedure itself can be passed around as arguments to other procedures. Procedures can be return values. They can also be stored in data structures. A similar corollary (though not exact) are composite functions which are usually taught in pre-calculus.
- Lisp programs can manipulate representations of Lisp programs - this likely refers to macros and how in Lisp code can be treated as data.

Core Terms/Concepts Learnt

- None

2 Preface

Key Takeaway: *The goal of the book is to teach the reader to think recursively.*

- Programs take data, apply a process on that data, and then produce some data.
- Recursion is the act of defining an object or solving a problem in terms of itself.
- The authors believe that writing programs recursively in Lisp is essentially pattern recongnition. Well I think it's true for any programming language or any programming paradigm.
- For recursive programming and studying, this book we will need only a few primitives/functions, namely:
  - `car`
  - `cdr`
  - `cons`
  - `atom?`
  - `eq?`
  - `null?`
  - `add1`, `sub1`
  - `and`, `or`
  - `else`
  - `lambda`
  - `cond`
  - `define`
- The definitions of the above primitives I am not outlining here and will come to it as we work through the book.
- Authors advise to read this book slowly. Very slowly and delibrately. Re-read it multiple times. Every concept should be clear before going onto the next page.

- In the preface we hit the first difference between Scheme and Common Lisp. `()` in Scheme is actually different from that in Common Lisp. Scheme considers `()` as *only* a list and *not* an atom. While in Common Lisp `()` is considered both an atom and a list. `ATOM` is defined as per the Lisp Hyperspec as well as Common Lisp The Language (2nd ed.) by Guy Steele as 'The predicate `ATOM` is true if its argument is not a `CONS`, and otherwise is false. In SBCL, a Common Lisp implementation `ATOM` will give `T`

  ```
  (atom '())
  » T
  ```

  We define our own predicate `atom?` in Scheme

  ```
  (define atom?
    (lambda (x)
      (and (not (pair? x)) (not (null? x)))))
  ```

  Some tests for checking `atom?` in Scheme below

  ```
  > (atom? 'a)
  #t
  > (atom? (quote ()))
  #f
  > (atom? '(a b c))
  #f
  > (atom? 42)
  #t
  ```

Core Terms/Concepts Learnt

- In Scheme `()` is only a list and not an atom.

3 Toys

This chapter introduces primitives of Scheme. These are the basic building blocks.

3.1 The Law of `car`

Key Takeaway: *The primitive* ***`car`*** *is defined only for non-empty lists. The* ***`car`*** *is the first atom (element) of that list.*

- An atom is indivisible - number, strings
- Anything enclosed in parenthesis/brackets `()` is a list.
- We can have nested lists which are also called improper lists and non-nested lists which are proper lists.
- An S-expression which stands for Symbolic Expression is any Lisp object that can be read and evaluated by the Lisp reader.
- Q. How many S-expressions are in the list `(how are you doing so far)` and what are they? The book answers 6 and those are the elements in the lists, basically the 6 atoms inside the list.
- A question asks how many S-expressions are in the list `(((how) are) ((you) (doing so)) far)` and gives the answer as 3. It refers to the 3 lists inside the outermost list.
- The difference of `()` again comes up since it is both a list and an atom in Common Lisp unlike Scheme. The `car` of `()` will be `NIL` in Common Lisp unlike Scheme. In Common Lisp as per the standards and empty list's `car` and `cdr` are both `NIL` (shown below).

```
(car ())
» NIL
```

- `car` is the first atom/element of a list. If we try to find the `car` of a string of character or numbers we will get an error like below.

```
  > (car 'a)
. . car: contract violation
  expected: pair?
  given: 'a
> (car 42)
. . car: contract violation
  expected: pair?
  given: 42
```

3.2 The Law of `cdr`

Key Takeaway: *The primitive* ***`cdr`*** *is defined only for non-empty lists. The* ***`cdr`*** *of any non-empty list is always another list.*

- The book says `car` of l is same as `(car l)`. Similarly for `cdr`.
- `cdr` of a single atom/element list is `()`.
- In Tourtezky's book there is a tool called SDRAW. It allows us to do draw `cons` cell structures with the `car` & `cdr` pointers. I have uploaded the code for this tool on Github here. For `(car a)` and `(cdr a)` where `a` is `samosa` will be represented as:

  ```
  [*|*]---> NIL
  |
  |
  V
  SAMOSA
  ```

- `cdr` of an empty list will be `()` as per Common Lisp standards but in Scheme it is an error.

3.3 The Law of `cons`

Key Takeaway: *The primitive* ***`cons`*** *takes two arguments. The second argument to* ***`cons`*** *must be a list. The result is a list.*

- `cons` actually creates a `cons` cell. The `car` of which is the first input to `cons` and the `cdr` is pointed to the second input. The return value of the `cons` is a pointer to it. Refer Touretzky's Chapter 2, clearly explained.

- Q. What is `(cons s l)` where `s` is `((a b c))` and `l` is `b`? This brings in the topic of Dotted Lists. In a proper list the chain of `cons` cells ends with `()` as the atom, meaning the last cell points to a `NIL` but in a dotted list the last atom points to a non `NIL` atom. In the above case we will get the following:

```
> (cons 'a '(b c))
'(a b c)
> (cons 'a 'b)
'(a . b)
```

3.4 The Law of `null?`

Key Takeaway: *The primitive **`null?`** is defined only for lists.*

- Q. Is it true that the list `l` is the null list where `l` is `()`? Yes, because it is composed of zero S-expressions.
- Another difference in Common Lisp and Scheme is how they refer to False. In scheme it is explicitly `#t` or `#f` but in Common Lisp it is `T` for True or else it is `NIL` which means False.
- `null?` of an atom should throw an error for a string or a number but actually it gives `#f` since in Scheme `null?` for `()` is `#t` and for everything else it is `#f`. See code below.

```
  > (null? 'a)
#f
> (null? (quote ()))
#t
```

3.5 The Law of `eq?`

Key Takeaway: *The primitive **`eq?`** in takes two arguments and compares them. Each must be a non numeric atom.*

- In `eq?` the address of the Lisp object is compared. For instance if we create two cons cells with same elements `eq?` will give `#f`

```
> (eq? (cons 'a 'b) (cons 'a 'b))
#f
```

Core Terms/Concepts Learnt

- `car`, `cdr`, `cons`, `eq?`, `null?`, `quote` / `'`, `#t`, `#f`,
- Atoms, S-Expressions, Lists, Dotted Lists

4 Do It, Do It Again, and Again, and Again ...

This chapter explains recursion. The best material for recursion in my opinion is Chapter 8 in Touretzky's book.

Key Takeaway: *The First Commandment (preliminary): Always ask* `null?` *as the first question in expressing a function*

- After reading Touretzky's chapter on recursion this chapter will feel very easy. Also the first commandment is not really true always. Sometimes in recursion the first question is not necessarily `null?`. Later in the book the authors do add in this nuance.
- The chapter introduces a function named `lat?`. It stands for a list of atoms. This means every element of the list is an atom. It can be written as below.

```
(define lat?
  (lambda (l)
    (cond
      ((null? l) #t)
      ((atom? (car l)) (lat? (cdr l)))
      (else #f))))
```

- It is important to understand how `cond` functions. Well `cond` is actually a macro. This macro has a series of tests and results. The macro goes from top to bottom. The cases are processed from left to right under each test. Technically we can have more than one result per test for evaluation. As a Lisp 'trick' the last test is usually an `else` which evaluates to `#t` always and hence the last result is returned. `cond` is a very nice way to implement `If..then..Else`. I have never seen such seamless conditional in any language yet.
- `lat?` basically is a `cond` which keeps checking through all the elements of a list to test for `atom?` till the list ends. It checks `car` one by one for each subsequent `cdr` for `atom?`.
- I would study Chapter 8 of Touretzky for getting the intuition on recursion right. The author has done a great job.

- Dr Racket comes with an inbuilt tool called `trace` in the library package called ~(racket/trace) which lets us see the actual function calls. So lets trace all the recursive examples in this chapter.

```
> (lat? '(Jack Sprat could eat no chicken fat))
>(lat? '(Jack Sprat could eat no chicken fat))
>(lat? '(Sprat could eat no chicken fat))
>(lat? '(could eat no chicken fat))
>(lat? '(eat no chicken fat))
>(lat? '(no chicken fat))
>(lat? '(chicken fat))
>(lat? '(fat))
>(lat? '())
<#t
#t
```

- Another example which has a nested list

```
> (lat? '(Jack (Sprat could) eat no chicken fat))
>(lat? '(Jack (Sprat could) eat no chicken fat))
>(lat? '((Sprat could) eat no chicken fat))
<#f
#f
```

- Few more examples from the chapter using `trace`

```
> (lat? '(bacon and eggs))
>(lat? '(bacon and eggs))
>(lat? '(and eggs))
>(lat? '(eggs))
>(lat? '())
<#t
#t
```

```
> (lat? '(bacon (and eggs)))
>(lat? '(bacon (and eggs)))
>(lat? '((and eggs)))
```

```
<#f
#f
```

- `or` is introduced as a logical operator. `or` asks two questions, one at a time. If the first one is true it stops answers true. Otherwise it asks the second question and answers with whatever the second question answers.

- `member?` is a function which returns a `#t` if the input is one of the elements in a list else `#f`. The book defines this function using `or` whereas it is actually not necessary.

```
    (define member?
      (lambda (a lat)
        (cond
          ((null? lat) #f)
          (else (or (eq? (car lat) a)
                    (member? a (cdr lat)))))))

(define my-member?
  (lambda (a lat)
    (cond
      ((null? lat) #f)
      ((eq? (car lat) a) #t)
      (else (member? a (cdr lat))))))
```

- The application of `member?` to find out whether *meat* is in the list *(mashed potatoes and meat gravy)* would generate this recursive call

```
> (member? 'meat '(mashed potatoes and meat gravy))
>(member? 'meat '(mashed potatoes and meat gravy))
>(member? 'meat '(potatoes and meat gravy))
>(member? 'meat '(and meat gravy))
>(member? 'meat '(meat gravy))
<#t
#t
```

- Another example

```
> (member? 'liver '(bagels and lox))
>(member? 'liver '(bagels and lox))
>(member? 'liver '(and lox))
>(member? 'liver '(lox))
>(member? 'liver '())
<#f
#f
```

Core Terms/Concepts Learnt

- `or`
- Basic template of recursion. Chapter 8 of Touretzky is great for a deeper dive into ways to construct recursion. Also tail optimized recursion is to be studied from the book 'Sketchy Scheme' by Nils M Holm.

5 Cons the Magnificent

This chapter explains the methods to build lists using `cons` recursively.

Key Takeaway: *The Second Commandment: Use* `cons` *to build lists The Third Commandment: When building a list, describe the first typical element, and then* `cons` *it onto the natural recursion*

- In last chapter we made a `member?` function and in this chapter we will be making a function which will *remove* a member
- The first attempt to build the `rember` function fails since it removes all the initial elements before finding the one it wants to remove. The authors have nicely demonstrated why `cons` is required to define this function.
- The way to write `rember` is as below. Also note as per Scheme semantics there is no `?` at the end of `rember` because it is actually not a predicate.

```
(define rember
  (lambda (a lat)
    (cond
      ((null? lat) (quote ()))
      ((eq? (car lat) a) (cdr lat))
      (else (cons (car lat)
                  (rember a (cdr lat)))))))
```

- There is a way to contrast the incorrect `rember` with the correct `cons` `rember` by looking at the recursive trace calls. The incorrect `rember-wrong` is below with its trace and return.

```
(define rember-wrong
  (lambda (a lat)
    (cond
      ((null? lat) #f)
      ((eq? (car lat) a) #t)
      (else (rember-wrong a (cdr lat))))))
```

```
> (rember-wrong 'and '(bacon lettuce and tomato))
>(rember-wrong 'and '(bacon lettuce and tomato))
>(rember-wrong 'and '(lettuce and tomato))
>(rember-wrong 'and '(and tomato))
<'(tomato)
'(tomato)
```

Whereas the correct trace and output is as below

```
> (rember 'and '(bacon lettuce and tomato))
>(rember 'and '(bacon lettuce and tomato))
> (rember 'and '(lettuce and tomato))
> >(rember 'and '(and tomato))
< <'(tomato)
< '(lettuce tomato)
<'(bacon lettuce tomato)
'(bacon lettuce tomato)
```

Another example

```
> (rember 'sauce '(soy sauce and tomato sauce))
>(rember 'sauce '(soy sauce and tomato sauce))
> (rember 'sauce '(sauce and tomato sauce))
< '(and tomato sauce)
<'(soy and tomato sauce)
'(soy and tomato sauce)
```

- Next function is `firsts` to build a list of the first S-expressions in nested lists. The code and trace is

```
(define firsts
  (lambda (l)
    (cond
      ((null? l) (quote ()))
      (else (cons (car (car l))
                  (firsts (cdr l)))))))
```

```
>(firsts
  '((apple peach pumpkin)
    (plum pear cherry)
    (grape raisin pea)
    (bean carrot eggplant)))
> (firsts '((plum pear cherry) (grape raisin pea) (bean carrot eggplant)))
> >(firsts '((grape raisin pea) (bean carrot eggplant)))
> > (firsts '((bean carrot eggplant)))
> > >(firsts '())
< < <'()
< < '(bean)
< <'(grape bean)
< '(plum grape bean)
<'(apple plum grape bean)
'(apple plum grape bean)
```

- The book refers to `seconds` but doesn't provide code for it. But I will try it out anyways.

```
(define seconds
  (lambda (l)
    (cond ((null? l) (quote ()))
          (else (cons (car (cdr (car l)))
                      (seconds (cdr l)))))))
```

```
>(seconds '((a b) (c d) (e f)))
> (seconds '((c d) (e f)))
> >(seconds '((e f)))
> > (seconds '())
< < '()
< <'(f)
< '(d f)
<'(b d f)
'(b d f)
```

- In the recursion technique of `cons`-ing cells the last `cons` cells' `cdr` pointer will point to a `nil` or an empty list `()`. Therefore, the usual terminal or base condition is to check for `null?` then `cons` a `()`.

- Although the book alludes to the fact that the `cons`-ing can be in any direction the trace in Dr Racket actually shows the `cons`-ing with a `<` or a `>`. So a `()` gets consed with the last recurring item and goes back to the first item. In the above example `()` is `cons`-ed to `f` to make a list `(f)`. This `(f)` is then `cons`-ed to `d` to make the list `(d f)`. This `(d f)` is then `cons`-ed to `b` to finally get `(b d f)`.

6 Numbers Games

7 * Oh My Gawd *: It's Full of Stars

8 Shadows

9 Friends and Relations

10 Lambda the Ultimate

11 ... and Again, and Again, and Again, ...

12 What Is the Value of All of This?

13 Intermission

14 The Ten Commandments

15 The Five Rules