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 Lisp. 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. But in Common Lisp the `CAR` of an empty list will give `NIL`.*

- In Common Lisp an `ATOM` is anything which is not a `CONS`.
- `ATOM` will include single characters, strings, numbers, special characters.
- 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. S-expressions include both `ATOM` and `CONS` (which is used to make lists).
- 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. But the list itself is an S-expression in Common Lisp so there are actually 7 S-expressions.
- The next 2 questions build up on this contradiction in my opinion. 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. So a list is an S-expression for this question but a list was not an S-expression for the prior question. Furthermore 3 should not be the correct answer here. The answer should be 12 in my opinion - 6 atoms (the words), 6 lists (nested and outermost).
- 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`.

```
(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 SBCL will give us a variable unbound error or say that the number is not of the type list.

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 `CDR` of an empty list in Common Lisp is `NIL`.*

- The book says `CAR` of l is same as `(car l)`. Similarly for `CDR`.
- `CDR` of a single atom/element list is `NIL` or `()`.
- 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 `NIL` as per Common Lisp standards.

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 NIL 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)) 'd)
» (((A B C)) . D)
```

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, but not because it is the list composed of zero S-expressions but because the list *contains* zero S-expressions. In Common Lisp we don't use ? at the end of predicates. So it is NULL in Common Lisp and null? in Scheme.

- 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. A section of notes on only NIL follows the end of this chapter.

```
(null '(gol gappa))
» NIL

(null '())
» T
```

- NULL of an atom will throw a variable unbound error for a string or say the number is not of the type list.

3.5 The Law of `EQ`

Key Takeaway: *The function `EQ` in Common Lisp takes two arguments and compares the unique address of these two arguments.*

- Now there are major difference here between the ways in which equality can be tested. First in Common Lisp symbols are unique where one symbol can have only address in computer's memory (within a given package in Common Lisp). This address of the symbol object is unique. So if we have a list as `(TIME AFTER TIME)` then `TIME` has the same address irrespective of the fact that it is repeated twice. The `EQ` function does this. This is a deviation from the `eq?` as defined in the book.

- Common Lisp has other equality tests
  - `EQ`: As explained above
  - `EQL`: Same as `EQ` but for two numbers it will compare the values. So integer 4 is different from a floating point 4.0
  - `EQUAL`: This compares the elements of a list one by one. It is slower than `EQ`. Most common testing method probably.
  - `EQUALP`: Same as `EQUAL` but ignores case
  - `=` Only for number comparison. Integer 4 and floating point 4.0 yields `T`

- We again come across the treatment of `()` differently in Scheme and Common Lisp. In Scheme `()` is only a list and not an atom but in Common Lisp it is both.

- One key point to note is that `EQ` is a function and not a primitive predicate in Common Lisp. We can verify this

  ```
  (functionp #'eq)
  » T
  ```

- The book states that two lists can be compared using `eq?`. In Common Lisp `EQ` function will indeed compare two lists but even if they contain same elements the lists could be distinct an thus `EQ` will return a `NIL`. Here if the intent is to compare the contents in the list then we need to use `EQUAL`.

```
(setf mithai (list 'ladoo 'barfi 'jalebi))
(setf sweets (list 'ladoo 'barfi 'jalebi))
(equal mithai sweets)
 ≫ T
(eq mithai sweets)
 ≫ NIL
```

- Numbers can also be compared using `EQ` since it will check their memory addresses.

3.6 Notes on `NIL` in Common Lisp

- Predicates are functions that answer questions in `T` or `NIL` (anything non-NIL is equivalent to `T`).
- `NIL` is the only way to say 'no' in Lisp. For instance the `NOT` predicate will return `NIL` for every input except `NIL` itself.
- A function is said to return 'false' when it returns `NIL`. But the function is said to return 'true' when it returns anything other than `NIL`.
- Anything other than `NIL` is treated as true in Lisp.
- A list of zero elements is called an empty list (do not use the term set). It has no cons cells. Denoted by empty brackets `()`.
- In the computer `()` i.e. empty list is represented by the symbol `NIL`.
- The symbol `NIL` is the empty list `()`. Thus, `NIL` is used to mark the end of a `CONS` cell chain.
- In bracket notation `NIL` at the end of the `CONS` chain is omitted as a convention.
- Since `NIL` and `()` are same they can be written interchangeably. Therefore, `(A () B)` is same as `(A NIL B)`
- The length of the empty list is 0. `NIL` can be passed to `LENGTH` since it is an empty list.
- `NIL` is the only thing which is a symbol and a list.

- The `CAR` and `CDR` of `NIL` is `NIL`.
- `NIL` like `T`, characters and numbers evaluates to itself. This is so because their value cells point to themselves.
- Explicit use as a symbol can be done by quoting i.e. `'NIL`
- Historically empty list was treated as false.

The 5 pointers of the symbol `NIL`

- Name: `NIL`
- Value: `NIL` (Thus it is self evaluating. The pointer from value of `NIL` goes back to the symbol `NIL` itself. Same with `T`)
- Function: No associated function
- plist: No associated properties
- Package: In Common Lisp Package

Core Terms/Concepts Learnt

- `CAR`, `CDR`, `CONS`, `EQ`, `NULL`, `NIL`
- 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`. Let's use an example. Make a recursive function to compute a factorial of a number.

  ```
  (defun fact (n)
    (cond ((zerop n) 1)
          (t (* n (fact (- n 1))))))
  ```

  Here the first question actually asks whether the argument `n` is a zero or not. 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?`. We will use our own defined `ATOMP` predicate so that we do not return `T` for a `()`.

- Recursive function definition of `LAT?` in Common Lisp.

  ```
  (defun lat? (l)
    (cond ((null l) t)
          ((atom? (car l)) (lat? (cdr l)))
          (t nil)))
  ```

- 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 Common Lisp 'trick' the last test is usually a `T` which evaluates to `TRUE` 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 `ATOMP`.

- I would study Chapter 8 of Touretzky for getting the intuition on recursion right. The author has done a great job.

- SBCL comes with an inbuilt tool called `TRACE` which lets us see the actual function calls. So lets trace all the recursive examples in this chapter.

```
CL-USER> (lat? '(Jack Sprat could eat no chicken fat))
  0: (LAT? (JACK SPRAT COULD EAT NO CHICKEN FAT))
    1: (LAT? (SPRAT COULD EAT NO CHICKEN FAT))
      2: (LAT? (COULD EAT NO CHICKEN FAT))
        3: (LAT? (EAT NO CHICKEN FAT))
          4: (LAT? (NO CHICKEN FAT))
            5: (LAT? (CHICKEN FAT))
              6: (LAT? (FAT))
                7: (LAT? NIL)
                7: LAT? returned T
              6: LAT? returned T
            5: LAT? returned T
          4: LAT? returned T
        3: LAT? returned T
      2: LAT? returned T
    1: LAT? returned T
  0: LAT? returned T
T
```

- Another example which has a nested list

```
CL-USER> (lat? '(Jack (Sprat could) eat no chickn fat))
0: (LAT? (JACK (SPRAT COULD) EAT NO CHICKN FAT))
  1: (LAT? ((SPRAT COULD) EAT NO CHICKN FAT))
  1: LAT? returned NIL
```

```
  0: LAT? returned NIL
NIL
```

- Few more examples from the chapter using `TRACE`

  ```
  CL-USER> (lat? '(bacon and eggs))
    0: (LAT? (BACON AND EGGS))
      1: (LAT? (AND EGGS))
        2: (LAT? (EGGS))
          3: (LAT? NIL)
          3: LAT? returned T
        2: LAT? returned T
      1: LAT? returned T
    0: LAT? returned T
  T


  CL-USER> (lat? '(bacon (and eggs)))
    0: (LAT? (BACON (AND EGGS)))
      1: (LAT? ((AND EGGS)))
      1: LAT? returned NIL
    0: LAT? returned NIL
  NIL
  ```

- `OR` is introduced as a logical operator. There is a very important point to note in the use of `OR` macro. Quoting from Chapter 4 in Touretzky - The rule for evaluating `OR`: Evaluate the clauses one at a time. If a clause returns something other than `NIL`, stop and return that value; otherwise go on to the next clause, or return `NIL` if none are left. `AND` is not introduced in this chapter. Quoting the evaluation rule for `AND` - Evaluate the clauses one at a time. If a clause returns `NIL`, stop and return `NIL`; otherwise go on to the next one. If all clauses yield non-NIL results, return the value of the last clause.

- `MEMBER` is a function which returns a `T` if the input is one of the elements in a list else `NIL~/#f`. The book defines this function using `OR` whereas it is actually not necessary.

```lisp
(defun my-member (a lat)
  (cond ((null lat) nil)
        ((equal a (car lat)) t)
        (t (my-member a (cdr lat)))))

(defun member? (a lat)
  (cond ((null lat) nil)
        (t (or (equal (car lat) a) (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

```
CL-USER> (member? 'meat '(mashed potatoes and meat gravy))
  0: (MEMBER? MEAT (MASHED POTATOES AND MEAT GRAVY))
    1: (MEMBER? MEAT (POTATOES AND MEAT GRAVY))
      2: (MEMBER? MEAT (AND MEAT GRAVY))
        3: (MEMBER? MEAT (MEAT GRAVY))
        3: MEMBER? returned T
      2: MEMBER? returned T
    1: MEMBER? returned T
  0: MEMBER? returned T
T
```

- Another example

```
CL-USER> (member? 'liver '(bagels and lox))
  0: (MEMBER? LIVER (BAGELS AND LOX))
    1: (MEMBER? LIVER (AND LOX))
      2: (MEMBER? LIVER (LOX))
        3: (MEMBER? LIVER NIL)
        3: MEMBER? returned NIL
      2: MEMBER? returned NIL
    1: MEMBER? returned NIL
  0: MEMBER? returned NIL
NIL
```

Core Terms/Concepts Learnt

- `OR`
- Basic template of recursion. Chapter 8 of Touretzky is great for a deeper dive into ways to constuct 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: *Use `CONS` to build lists*

- 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` in Common Lisp is as below. Also note as per Scheme semantics there is no `?` at the end of `REMBER` because it is actually not a predicate.

```
(defun rember (a lat)
  (cond ((null lat) nil)
        ((equal a (car lat)) (cdr lat))
        (t (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.

```
    (defun rember-wrong (a lat)
      (cond ((null lat) nil)
            ((equal a (car lat)) (cdr lat))
            (t (rember-wrong a (cdr lat)))))
CL-USER> (rember-wrong 'and '(bacon lettuce and tomato))
(TOMATO)

CL-USER> (rember-wrong 'and '(bacon lettuce and tomato))
  0: (REMBER-WRONG AND (BACON LETTUCE AND TOMATO))
    1: (REMBER-WRONG AND (LETTUCE AND TOMATO))
      2: (REMBER-WRONG AND (AND TOMATO))
```

```
      2: REMBER-WRONG returned (TOMATO)
    1: REMBER-WRONG returned (TOMATO)
  0: REMBER-WRONG returned (TOMATO)
(TOMATO)
```

Whereas the correct trace and output is as below

```
CL-USER> (rember 'and '(bacon lettuce and tomato))
(BACON LETTUCE TOMATO)

CL-USER> (rember 'and '(bacon lettuce and tomato))
  0: (REMBER AND (BACON LETTUCE AND TOMATO))
    1: (REMBER AND (LETTUCE AND TOMATO))
      2: (REMBER AND (AND TOMATO))
      2: REMBER returned (TOMATO)
    1: REMBER returned (LETTUCE TOMATO)
  0: REMBER returned (BACON LETTUCE TOMATO)
(BACON LETTUCE TOMATO)
```

Another example

```
CL-USER> (rember 'sauce '(soy sauce and tomato sauce))
  0: (REMBER SAUCE (SOY SAUCE AND TOMATO SAUCE))
    1: (REMBER SAUCE (SAUCE AND TOMATO SAUCE))
    1: REMBER returned (AND TOMATO SAUCE)
  0: REMBER returned (SOY AND TOMATO SAUCE)
(SOY AND TOMATO SAUCE)
```

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