The Little Schemer Simplified

Deepak Venkatesh

November 1, 2025

Abstract

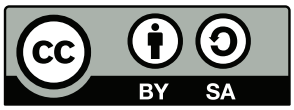
These are my personal notes created to deepen my understanding of Lisp and programming in general. *The Little Schemer* (4th edition) by Daniel Friedman and Matthias Felleisen is a remarkable book that teaches programming concepts in a unique and playful way. It builds from first principles using only a small set of primitives, showing how powerful ideas — such as recursion, functional programming, lambda functions, closures, higher order functions, and interpreters — can be expressed using just those few building blocks. I am sharing this 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



Contents

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 below (doesn't render well in markdown file on github):

  ```
  [*|*]---> 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 The Fourth Commandement: Always change at lest one argument while recurring. It must be changed to be closer to termination. The changing argument must be tested in the termination condition: when using* ***`cdr`****, test termination with* ***`null?`****.*

- 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 `cons`-ed 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)`.

- Without looking into the questions was able to build these `cons`-es for `insertR`, `insertL`, and `subst`.

```
(define insertR
  (lambda (new old lat)
    (cond ((null? lat) (quote ()))
          ((eq? old (car lat)) (cons (car lat) (cons new (cdr lat))))
          (else (cons (car lat) (insertR new old (cdr lat)))))))

(define insertL
  (lambda (new old lat)
    (cond ((null? lat) (quote ()))
          ((eq? old (car lat)) (cons new (cons (car lat) (cdr lat))))
          (else (cons (car lat) (insertL new old (cdr lat)))))))

(define subst
  (lambda (new old lat)
    (cond ((null? lat) (quote ()))
          ((eq? old (car lat)) (cons new (cdr lat)))
          (else (cons (car lat) (subst new old (cdr lat)))))))
```

- `subst2` is another function defined in the book, it substitutes either the first occurrence of one thing or another.

```
(define subst2
  (lambda (new o1 o2 lat)
    (cond ((null? lat) (quote ()))
          ((or (eq? (car lat) o1) (eq? (car lat) o2)) (cons new (cdr lat)))
          (else (cons (car lat) (subst2 new o1 o2 (cdr lat)))))))
```

- `subst2` involves an `or` but a better example could be used such as below. Since trace will give a better view.

```
(define subst2
  (lambda (new o1 o2 lat)
    (cond ((null? lat) (quote ()))
          ((or (eq? (car lat) o1) (eq? (car lat) o2)) (cons new (cdr lat)))
          (else (cons (car lat) (subst2 new o1 o2 (cdr lat)))))))
```

```
> (subst2 'vanilla 'chocolate 'banana
          '(caramel raspberry ice cream with
                    chocolate topping with some bananas))
>(subst2
  'vanilla
  'chocolate
  'banana
  '(caramel raspberry ice cream with chocolate topping with some bananas))
> (subst2
   'vanilla
   'chocolate
   'banana
   '(raspberry ice cream with chocolate topping with some bananas))
> >(subst2
    'vanilla
    'chocolate
    'banana
    '(ice cream with chocolate topping with some bananas))
> > (subst2
     'vanilla
     'chocolate
     'banana
     '(cream with chocolate topping with some bananas))
> > >(subst2
      'vanilla
      'chocolate
      'banana
      '(with chocolate topping with some bananas))
> > > (subst2
       'vanilla
       'chocolate
```

```
        'banana
        '(chocolate topping with some bananas))
< < < '(vanilla topping with some bananas)
< < <'(with vanilla topping with some bananas)
< < '(cream with vanilla topping with some bananas)
< <'(ice cream with vanilla topping with some bananas)
< '(raspberry ice cream with vanilla topping with some bananas)
<'(caramel raspberry ice cream with vanilla topping with some bananas)
'(caramel raspberry ice cream with vanilla topping with some bananas)
```

- `multirember` is below. I am unsure why two `else` is used by the author. Probably I will learn it down the line. But the `trace` captures the series of recursive steps beautifully. Note the > and < they capture the stack trace nicely.

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

```
> (multirember 'cup '(coffee cup tea cup and hick cup))
>(multirember 'cup '(coffee cup tea cup and hick cup))
> (multirember 'cup '(cup tea cup and hick cup))
> (multirember 'cup '(tea cup and hick cup))
> >(multirember 'cup '(cup and hick cup))
> >(multirember 'cup '(and hick cup))
> > (multirember 'cup '(hick cup))
> > >(multirember 'cup '(cup))
> > >(multirember 'cup '())
< < <'()
< < '(hick)
< <'(and hick)
< '(tea and hick)
<'(coffee tea and hick)
'(coffee tea and hick)
```

- Function definition for other 'multi' variants below: `multiinsertR`, `multiinsertL`, and `multisubst`.

```
    (define multiinsertR
      (lambda (new old lat)
        (cond ((null? lat) (quote ()))
              ((eq? old (car lat))
               (cons (car lat)(cons new (multiinsertR new old (cdr lat)))))
              (else (cons (car lat) (multiinsertR new old (cdr lat)))))))

> (multiinsertR 'bag 'cup '(coffee cup tea cup and hick cup))
>(multiinsertR 'bag 'cup '(coffee cup tea cup and hick cup))
> (multiinsertR 'bag 'cup '(cup tea cup and hick cup))
> >(multiinsertR 'bag 'cup '(tea cup and hick cup))
> > (multiinsertR 'bag 'cup '(cup and hick cup))
> > >(multiinsertR 'bag 'cup '(and hick cup))
> > > (multiinsertR 'bag 'cup '(hick cup))
> > > >(multiinsertR 'bag 'cup '(cup))
> > > > (multiinsertR 'bag 'cup '())
< < < < '()
< < < <'(cup bag)
< < < '(hick cup bag)
< < <'(and hick cup bag)
< < '(cup bag and hick cup bag)
< <'(tea cup bag and hick cup bag)
< '(cup bag tea cup bag and hick cup bag)
<'(coffee cup bag tea cup bag and hick cup bag)
'(coffee cup bag tea cup bag and hick cup bag)


    (define multiinsertL
      (lambda (new old lat)
        (cond ((null? lat) (quote ()))
              ((eq? old (car lat))
               (cons new (cons (car lat) (multiinsertL new old (cdr lat)))))
              (else (cons (car lat) (multiinsertL new old (cdr lat)))))))

    > (multiinsertL 'bag 'cup '(coffee cup tea cup and hick cup))
>(multiinsertL 'bag 'cup '(coffee cup tea cup and hick cup))
> (multiinsertL 'bag 'cup '(cup tea cup and hick cup))
> >(multiinsertL 'bag 'cup '(tea cup and hick cup))
> > (multiinsertL 'bag 'cup '(cup and hick cup))
> > >(multiinsertL 'bag 'cup '(and hick cup))
```

```
> > > (multiinsertL 'bag 'cup '(hick cup))
> > > >(multiinsertL 'bag 'cup '(cup))
> > > > (multiinsertL 'bag 'cup '())
< < < < '()
< < < <'(bag cup)
< < < '(hick bag cup)
< < <'(and hick bag cup)
< < '(bag cup and hick bag cup)
< <'(tea bag cup and hick bag cup)
< '(bag cup tea bag cup and hick bag cup)
<'(coffee bag cup tea bag cup and hick bag cup)
'(coffee bag cup tea bag cup and hick bag cup)
```

```
    (define multisubst
      (lambda (new old lat)
        (cond ((null? lat) (quote ()))
              ((eq? (car lat) old) (cons new (multisubst new old (cdr lat))))
              (else (cons (car lat) (multisubst new old (cdr lat)))))))
```

```
> (multisubst 'bag 'cup '(coffee cup tea cup and hick cup))
>(multisubst 'bag 'cup '(coffee cup tea cup and hick cup))
> (multisubst 'bag 'cup '(cup tea cup and hick cup))
> >(multisubst 'bag 'cup '(tea cup and hick cup))
> > (multisubst 'bag 'cup '(cup and hick cup))
> > >(multisubst 'bag 'cup '(and hick cup))
> > > (multisubst 'bag 'cup '(hick cup))
> > > >(multisubst 'bag 'cup '(cup))
> > > > (multisubst 'bag 'cup '())
< < < < '()
< < < <'(bag)
< < < '(hick bag)
< < <'(and hick bag)
< < '(bag and hick bag)
< <'(tea bag and hick bag)
< '(bag tea bag and hick bag)
<'(coffee bag tea bag and hick bag)
'(coffee bag tea bag and hick bag)
```

Core Terms/Concepts Learnt

- Method to generate lists by using `cons` in recursions.

6 Numbers Games

This chapter explains how numbers can be built with recursion.

Key Takeaway:
The First Commandment (first revision): When recurring on a list of atoms, `lat`*, asks two questions about it:* `(null? lat)` *and* `else`*. When recurring on a number,* `n`*, ask two questions about it:* `(zero? n)` *and* `else`*.*

The Fourth Commandment (first revision): Always change one argument while recurring. It must be changed closer to termination. The changing argument must be tested in the termination condition: when using `cdr` *test termination with null? and when using* `sub1`*, test termination with* `zero?`*.*

The Fifth Commandment: When building a value with `+`*, always use 0 for the value of the terminating line, for adding 0 does not change the value of an addition. When building a value with* `x`*, always use 1 for the value of the terminating line, for multiplying by 1 does not change the value of a multiplication. When building a value with* `cons`*, always consider* `()` *for the value of the terminating line.*

- In this chapter we are focusing on only the elements in the set of whole numbers.
- We define the basic primitive functions to add 1 or subtract 1. Using this increment or decrement my assumption is we will create the Whole number set.

```
(define add1
  (lambda (n)
    (+ n 1)))

(define sub1
  (lambda (n)
    (- n 1)))
```

  If we do `(sub1 0)` we will actually get `-1` but for the sake of the book we will deal only with non-negative integers.

- `zero?` is an inbuilt predicate just like `ZEROP` in Common Lisp.

- We define addition of two numbers by making a decrementing counter of one of the numbers till it reaches zero. For every decrement we `add1` to the other number. So for instance we need to add 3 to 2 then the 3 goes to 2 then 1 then 0. So there are three steps 3 to 2, 2 to 1 and 1 to 0. So these three steps gets added to 2. Thus we get 5. Better to look at the stack trace for the example below. We are using the letter `o` to denote that its our 'own' definition.

```
(define o+
  (lambda (n m)
    (cond ((zero? m) n)
          (else (add1 (o+ n (sub1 m)))))))
```

```
> (o+ 2 3)
>(o+ 2 3)
> (o+ 2 2)
> >(o+ 2 1)
> > (o+ 2 0)
< < 2
< <3
< 4
<5
5
```

- In the earlier chapter the authors had referred to using `(null?)` as the first test and now they correctly say that for numbers we can use `zero?` as the test.
- `zero?` is like `null?` and `add1` is like `cons`
- Exactly like `o+` we can build `o-` . The only difference is that we do not `add1` on every decrement but rather subtract using `sub1`. Looking at the stack trace below.

```
(define o-
  (lambda (n m)
    (cond ((zero? m) n)
          (else (sub1 (o- n (sub1 m)))))))
```

```
> (o- 5 3)
>(o- 5 3)
> (o- 5 2)
> >(o- 5 1)
> > (o- 5 0)
< < 5
< <4
< 3
<2
2
```

- Tuple is defined as a list of numbers. In this case I am assuming non negative numbers and also the book say an empty list will be also a tuple. I don't think tuple is defined formally in the R^{5}RS standard.
- `()` is also a tuple as it is a list of zero numbers.
- `addtup` is essentially a function which does digit-sum (sums the numbers in the tuple).
- We will use `o+` to build numbers just like `cons` is used to build lists.
- Writing `addtup` seems easy given the exercises prior to this and reading Touretzky. Side comment: This book is actually really fun!

```
(define addtup
  (lambda (tup)
    (cond ((null? tup) 0)
          (else (o+ (car tup) (addtup (cdr tup)))))))
```

```
> (addtup '(1 2 3 4 5 6 7 8 9 10))
>(addtup '(1 2 3 4 5 6 7 8 9 10))
> (addtup '(2 3 4 5 6 7 8 9 10))
> >(addtup '(3 4 5 6 7 8 9 10))
> > (addtup '(4 5 6 7 8 9 10))
> > >(addtup '(5 6 7 8 9 10))
> > > (addtup '(6 7 8 9 10))
> > > >(addtup '(7 8 9 10))
> > > > (addtup '(8 9 10))
```

```
> > > > >(addtup '(9 10))
> > > > > (addtup '(10))
> > > >[10] (addtup '())
< < < <[10] 0
< < < < < 10
< < < < <19
< < < < 27
< < < <34
< < < 40
< < <45
< < 49
< <52
< 54
<55
55
```

- Multiplication is repetitive addition. So to build `x` we have to decrement one number and for every decrement add the other number to itself.

```
(define x
  (lambda (n m)
    (cond ((zero? m) 0)
          (else (o+ n (x n (sub1 m)))))))
```

```
> (x 4 3)
>(x 4 3)
> (x 4 2)
> >(x 4 1)
> > (x 4 0)
< < 0
< <4
< 8
<12
12
```

- A nice expansion in the book is for `(x 12 3)` fairly similar to the `trace` Dr Racket generates

```
(x 12 3)
= 12 + (x 12 2)
= 12 + 12 + (x 12 1)
= 12 + 12 + 12 + (x 12 0)
= 12 + 12 + 12 + 0
= 12 + 24
= 36
```

- A question is asked why is 0 the value for the terminal condition line in `x` and the answer to this is because 0 will not affect `+`. That is `n + 0 = n`. The actual math behind lies in abstract algebra. In an operation such as `+` there is a concept of identity and inverse. The `identity` or `neutral` element in the set of this operation does not affect the value of other elements when the operation is applied between an element and this identity. For example, in the operation of `+` the `identity` element is `0`. The operation `+` say is applied to the set of non-negative numbers (as done in this book). So `+ 2` and the `identity` should yield 2 itself. Thus the identity in this set of whole numbers for this specific `+` operation is 0. Similarly for the operation of `x` in the set of natural numbers the identity is 1. 2 multiplied by 1 yields 2 again. Now we get back to scheme and away from abstract algebra.

- The next function we write is addition of two tuples. In this all elements in the tuple at their respective positions are added. The first version of the code adds two tuples of the same length (code is below). When we supply it with varying length tuples we get an error because it tries to add a number to an empty list. The trace diagram shows the error. Now we will write a cleaner function which will take varying length tuples.

```
; this v1 version will work only if length of tup1 and tup2 is same
    (define tup+v1
      (lambda (tup1 tup2)
        (cond ((and (null? tup1) (null? tup2)) (quote ()))
              (else (cons (o+ (car tup1) (car tup2))
                          (tup+v1 (cdr tup1) (cdr tup2)))))))

> (tup+v1 '(1 2 3 4) '(4 3 2 1))
>(tup+v1 '(1 2 3 4) '(4 3 2 1))
```

```
> (tup+v1 '(2 3 4) '(3 2 1))
> >(tup+v1 '(3 4) '(2 1))
> > (tup+v1 '(4) '(1))
> > >(tup+v1 '() '())
< < <'()
< < '(5)
< <'(5 5)
< '(5 5 5)
<'(5 5 5 5)
'(5 5 5 5)

> (tup+v1 '(1 2 3 4) '(4 3 2))
>(tup+v1 '(1 2 3 4) '(4 3 2))
> (tup+v1 '(2 3 4) '(3 2))
> >(tup+v1 '(3 4) '(2))
> > (tup+v1 '(4) '())
. . car: contract violation
  expected: pair?
  given: '()
```

- `tup+` below is the correct way to define addition of the elements of two tuples. The trace diagram helps understand why. When one of the tuples runs out of elements i.e. it is an empty tuple, then at that time whatever the present recurring state of the other tuple is that is used for `cons`-ing when the stack frames start returning values.

```
(define tup+
  (lambda (tup1 tup2)
    (cond ((null? tup1) tup2)
          ((null? tup2) tup1)
          (else (cons (o+ (car tup1) (car tup2))
                      (tup+ (cdr tup1) (cdr tup2)))))))
```

```
> (tup+ '(1 2 3 4) '(4 3 2 1))
>(tup+ '(1 2 3 4) '(4 3 2 1))
> (tup+ '(2 3 4) '(3 2 1))
> >(tup+ '(3 4) '(2 1))
> > (tup+ '(4) '(1))
> > >(tup+ '() '())
```

```
< < <'()
< < '(5)
< <'(5 5)
< '(5 5 5)
<'(5 5 5 5)
'(5 5 5 5)

> (tup+ '(1 2 3 ) '(4 3 2 1))
>(tup+ '(1 2 3) '(4 3 2 1))
> (tup+ '(2 3) '(3 2 1))
> >(tup+ '(3) '(2 1))
> > (tup+ '() '(1))
< < '(1)
< <'(5 1)
< '(5 5 1)
<'(5 5 5 1)
'(5 5 5 1)

> (tup+ '(1 2 3 4) '(4 3 2))
>(tup+ '(1 2 3 4) '(4 3 2))
> (tup+ '(2 3 4) '(3 2))
> >(tup+ '(3 4) '(2))
> > (tup+ '(4) '())
< < '(4)
< <'(5 4)
< '(5 5 4)
<'(5 5 5 4)
'(5 5 5 4)
```

- Definitions of greater than and smaller than `>` and `<` is tricky. For greater than `>` the order of tests matter. The `#f` needs to be tested first for the base condition. This is so because when `n` reaches zero we know for sure that `n` is `<=` `m` thus the overall test is `#f`. But if we had tested `m` as zero which would return `#t` even if the condition `=` is satisfied. Similarly we can compose a function for lesser than `<`

```
(define >
  (lambda (n m)
    (cond ((zero? n) #f)
```

```
              ((zero? m) #t)
              (else (> (sub1 n) (sub1 m))))))
> (> 3 1)
>(> 3 1)
>(> 2 0)
<#t
#t

> (> 1 4)
>(> 1 4)
>(> 0 3)
<#f
#f

> (> 4 4)
>(> 4 4)
>(> 3 3)
>(> 2 2)
>(> 1 1)
>(> 0 0)
<#f
#f


    (define <
      (lambda (n m)
        (cond ((zero? m) #f)
              ((zero? n) #t)
              (else (< (sub1 n) (sub1 m))))))

> (< 1 4)
>(< 1 4)
>(< 0 3)
<#t
#t
> (< 4 1)
>(< 4 1)
>(< 3 0)
<#f
#f
```

```
> (< 4 4)
>(< 4 4)
>(< 3 3)
>(< 2 2)
>(< 1 1)
>(< 0 0)
<#f
#f
```

- Next we compose the equality function = for numbers. The plain vanilla method is to check if one is zero and whether at the same time the other is zero too. If not then it is false. Also if one reaches zero while decrementing and the other is still not zero then it is not equal. The code is simple below. The other way which build up on > and < is to check if these two are false then = will be true.

```
(define o=
  (lambda (n m)
    (cond ((zero? m) (zero? n))
          ((zero? n) #f)
          (else (= (sub1 n) (sub1 m))))))

(define =
  (lambda (n m)
    (cond ((> n m) #f)
          ((< n m) #f)
          (else #t))))
```

- Exponents (or raising to power) is also simple.

```
    (define o-exp
      (lambda (n m)
        (cond ((zero? m) 1)
              (else (x n (o-exp n (sub1 m)))))))

> (o-exp 2 4)
>(o-exp 2 4)
> (o-exp 2 3)
```

```
> >(o-exp 2 2)
> > (o-exp 2 1)
> > >(o-exp 2 0)
< < <1
< < 2
< <4
< 8
<16
16
```

- Integer division is implemented smartly by basically figuring out the how many wholes of a number fits into another. Thus discarding any of the remainder. Wonder how remainder can be obtained via recursion especially for recurring decimals or if I throw an irrational number.

```
    (define o-div
      (lambda (n m)
        (cond ((< n m) 0)
              (else (add1 (o-div (o- n m) m))))))
> (o-div 16 3)
>(o-div 16 3)
> (o-div 13 3)
> >(o-div 10 3)
> > (o-div 7 3)
> > >(o-div 4 3)
> > > (o-div 1 3)
< < < 0
< < <1
< < 2
< <3
< 4
<5
5
```

- In Common Lisp `LENGTH` is provided in the common user package as a function but here in Scheme we have to build it ourselves. Its quite easy.

```
    (define length
```

```
      (lambda (lat)
        (cond ((null? lat) 0)
              (else (add1 (length (cdr lat)))))))

> (length '(gulab jamun ladoo jalebi))
>(length '(gulab jamun ladoo jalebi))
> (length '(jamun ladoo jalebi))
> >(length '(ladoo jalebi))
> > (length '(jalebi))
> > >(length '())
< < <0
< < 1
< <2
< 3
<4
4
```

- Indexing a list or like in Python we refer to elements of a list starting with 0. In this example the book starts indexing with 1. Here I show both indexes one starting with 0 and the other as in the book starting with 1.

```
(define pick-zero
  (lambda (n lat)
    (cond ((zero? n) (car lat))
          (else (pick-zero (sub1 n) (cdr lat))))))

(define pick
  (lambda (n lat)
    (cond ((zero? (sub1 n)) (car lat))
          (else (pick (sub1 n) (cdr lat))))))

    > (pick-zero 2 '(paneer butter masala curry))
    >(pick-zero 2 '(paneer butter masala curry))
    >(pick-zero 1 '(butter masala curry))
    >(pick-zero 0 '(masala curry))
    <'masala
    'masala
```

```
> (pick 2 '(paneer butter masala curry))
>(pick 2 '(paneer butter masala curry))
>(pick 1 '(butter masala curry))
<'butter
'butter
```

- Removing an element in the list is consing like we did before.

```
(define rempick
  (lambda (n lat)
    (cond
      ((zero? (sub1 n)) (cdr lat))
      (else (cons (car lat)
                  (rempick (sub1 n) (cdr lat)))))))
```

- `number?` is a primitive in Scheme just like `NUMERBP` in Common Lisp.

```
> (number? 5)
#t
> (number? 'dosa)
#f
```

- Removing all numbers from a given list is a combination of using `number?` and a `cons` recursion. Unsure why the authors use two `else`'s. It makes the function long and a tad bit complicated.

```
(define no-nums
  (lambda (lat)
    (cond
      ((null? lat) (quote ()))
      ((number? (car lat)) (no-nums (cdr lat)))
      (else (cons (car lat) (no-nums (cdr lat)))))))

(define no-nums-else
  (lambda (lat)
    (cond
      ((null? lat) (quote ()))
      (else (cond
```

```
((number? (car lat))
 (no-nums-else (cdr lat)))
(else (cons (car lat)
            (no-nums-else (cdr lat)))))))))
```

- The next function is inverse of the former `no-nums`. In this `all-nums` we keep only the numbers and do away with everything else. This is also simple.

```
(define all-nums
  (lambda (lat)
    (cond
      ((null? lat) (quote ()))
      ((number? (car lat)) (cons
                            (car lat) (all-nums (cdr lat))))
      (else (all-nums (cdr lat))))))
```

- In Touretzky's book on Common Lisp there is a very nice explanation of equality tests such as `EQUAL`, `=`, `EQ` etc. Similarly in Scheme there are differences in the way equality tests are done. The table below captures the notes on this topic.

Predicate	Meaning	Works On
`eq?`	Object identity - Are these the same objects?	symbols, booleans
`eqv?`	Value identity (atomic) - Do these denote the same values?	numbers, chars
`equal?`	Structural equality - Are these structurally the same?	lists, strings, etc.
`=`	Numeric equality - Are these numbers same?	numbers

Predicate	Example
`eq?`	(eq? 'a 'a) → #t
`eqv?`	(eqv? 3 3) → #t
`equal?`	(equal? '(a b) '(a b)) → #t
`=`	(= 3 3.0) → #t

My assumption is that the *object* refers to the Lisp object therefore it is that specific location in memory which is being compared. Essentially are these two things in the same memory location? I will need to confirm this. Rest of the equality predicates are straightforward. Test to check the Lisp Object hypothesis:

```
> (eq? '(a b) '(a b))
#f
> (equal? '(a b) '(a b))
#t
```

- Next we write a function to check if the atoms are same. Here we use `=` for comparing numbers, and rightly so.

```
(define eqan?
  (lambda (a1 a2)
    (cond
      ((and (number? a1) (number? a2))
       (= a1 a2))
      ((or (number? a1) (number? a2)) #f)
      (else (eq? a1 a2)))))
```

- Not sure of the answer to the question "Can we assume that all functions written using `eq?` can be generalized by replacing `eq?` by `eqan?`". The answer given is "Yes" except for `equan?` itself. Is it because we can't write `equan?` without `eq?`. Not sure if I understood this correctly.

- `occur` is defined which basically counts the frequency of occurrence in a list.

```
(define occur
  (lambda (a lat)
    (cond
      ((null? lat) 0)
      ((eq? a (car lat)) (add1 (occur a (cdr lat))))
      (else (occur a (cdr lat))))))
```

- The chapter suddenly jumps to defining `one?` predicate which essentially checks if a number is `1` or not. We do it in multiple ways but we learn that `cond` is not necessary to construct this predicate.

```
(define one-zero?
  (lambda (n)
    (cond
      ((zero? n) #f)
      (else (zero? (sub1 n))))))

(define one-equal?
  (lambda (n)
    (cond
      (else (= n 1)))))

(define one?
  (lambda (n)
    (= n 1)))
```

- The last function of the chapter is `rempick` using `one?`. This is just a small change to our original `rempick` function.

```
(define rempick-one
  (lambda (n lat)
    (cond
      ((one? n) (cdr lat))
      (else (cons (car lat)
                  (rempick-one (sub1 n) (cdr lat)))))))
```

Core Terms/Concepts Learnt

- Building numbers by adding or subtracting 1
- Building primitive arithmetic operations of `+`, `-`, `x` and `/` (integer)
- Operations on tuples
- Different ways to do equality testing and counting frequency of occurrence

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

This chapter we start looking at nested lists.

Key Takeaway: *To be done*

- Removing a member in a nested list. In earlier chapters we only focused on proper non-nested lists but in this we will look at lists within lists. There could be any level of nesting theoretically speaking.
- Any function name with a `*` at the end basically means application for the entire list. We write `rember*` which removes a specific element from even nested lists. The logic is to simply check is the element in the list is an atom and equal to the removable element, if it is then we just do a recursive `rember*` on the `cdr` of the list. If it is not equal then we `cons` that to the recursive call of the `cdr` of the list. In the case where the element is a list itself we keep calling the list recursively that is the `car` and applying the `rember*`. It is fairly logical. The `trace` on the stack actually shows the entire process nicely.

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

```
> (rember* 'cup '((coffee) cup ((tea) cup)
                           (and (hick)) cup))
>(rember* 'cup '((coffee) cup ((tea) cup) (and (hick)) cup))
> (rember* 'cup '(coffee))
> >(rember* 'cup '())
< <'()
< '(coffee)
> (rember* 'cup '(cup ((tea) cup) (and (hick)) cup))
> (rember* 'cup '(((tea) cup) (and (hick)) cup))
> >(rember* 'cup '((tea) cup))
> > (rember* 'cup '(tea))
```

```
> > >(rember* 'cup '())
< < <'()
< < '(tea)
> > (rember* 'cup '(cup))
> > (rember* 'cup '())
< < '()
< <'((tea))
> >(rember* 'cup '((and (hick)) cup))
> > (rember* 'cup '(and (hick)))
> > >(rember* 'cup '((hick)))
> > > (rember* 'cup '(hick))
> > > >(rember* 'cup '())
< < < <'()
< < < '(hick)
> > > (rember* 'cup '())
< < < '()
< < <'((hick))
< < '(and (hick))
> > (rember* 'cup '(cup))
> > (rember* 'cup '())
< < '()
< <'((and (hick)))
< '(((tea)) (and (hick)))
<'((coffee) ((tea)) (and (hick)))
'((coffee) ((tea)) (and (hick)))
```

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