"Programs must be written for people to read,
and only incidentally for machines to execute." [1]

[1] Structure & Interpretation of Computer Programs by Abelson, Sussman & Sussman

The Schemer Simplified

Deepak Venkatesh

November 22, 2025

Abstract

This is fundamentally a study guide from my personal notes initially 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 pages. As Richard Feynman had said 'If you want to master something, teach it.' In the future I hope to append a similar guide for the sequel 'The Seasoned Schemer' to this book.

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. Throughout this guide I have given additional clarification and background about computer science concepts which are not present in the 'The Little Schemer'.

I am really thankful to Alonzo Church, John McCarthy, Daniel Friedman, Matthias Felleisen, Gerald Sussman, Guy Steele, and the PLT research group for giving to us this wonderful language and book. I stumbled upon Lisp very late in life via a blog by Steve Losh and ventured into the grand language of Common Lisp via David Touretzky, Peter Seibel, Paul Graham, and Peter Norvig. As a non programmer whose daily job has never involved writing a line of production code I

could not have spent hours on end without the support of my family. My daughter knows the names of half a dozen programming languages by now and has tried her hand at some concepts. She prefers Python though.

Who is this book and guide for? At the end of The Seasoned Schemer the authors say 'You have reached the end of your introduction to computation.' But I feel differently about it. This book is not an introduction to programming and neither this guide is. Readers should either have some understanding of Lisp to work through this book or experience with programming. If a reader does take this book on without any knowledge of programming and a bit of introduction to Lisp then they will find it difficult but not impossible to work through especially the chapters 8, 9, and 10 in the book. Chapter 9 is the most difficult followed by chapter 8 and then chapter 10. Rest of the chapters are fairly easy to moderate. I would say chapter 9 is the most difficult in both the books. I will be adding more details and illustrations at a future date to make this even more readable.

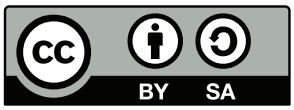
This guide is not intended for any commercial purpose and all rights of 'The Little Schemer' and 'The Seasoned Schemer' are with the respective authors and publishers. This is only a deep study of these two wonderful books.

All mistakes in this guide, whether typographical or conceptual, are entirely my own. Any misinterpretations are as well. I am still learning Lisp and programming. With every subsequent reading of this book I learn something new and peel another layer of Lisp.

Hardware and Software used for this study

- Language: Racket (a Scheme)
- Editor: Dr Racket
- emacs for Org notes
- chatgpt for assistance on Git and Org mode
- Macbook Pro 2019 2.6 Ghz I7 Intel chip with 16 GB RAM

'(deepak venkatesh)
'(redmond wa usa)



Contents

1 What is Lisp? 8
- 1.1 History . . . 8
- 1.2 Why Lisp? . . . 8
- 1.3 Resources to learn Lisp . . . 9

2 Foreword 10

3 Preface 11

4 The Five Rules 13

5 The Ten Commandments 14

6 Toys 16
- 6.1 The Law of `car` . . . 16
- 6.2 The Law of `cdr` . . . 17
- 6.3 The Law of `cons` . . . 17
- 6.4 The Law of `null?` . . . 18
- 6.5 The Law of `eq?` . . . 18

7 Do It, Do It Again, and Again, and Again ... 20
- 7.1 Basics of Recursion . . . 20
  - 7.1.1 The important `cond` . . . 20
  - 7.1.2 Trace tool in Dr Racket . . . 21

8 Cons the Magnificent 24
- 8.1 List Surgery . . . 24
  - 8.1.1 Removing elements . . . 24
  - 8.1.2 Finding elements . . . 25
  - 8.1.3 Inserting and substituting elements . . . 27
  - 8.1.4 Multiple surgeries on lists . . . 30

9 Numbers Games 33
- 9.1 Addition and Subtraction . . . 33
- 9.2 Tuples . . . 35
- 9.3 Multiplication . . . 36
  - 9.3.1 Digression to Abstract Algebra . . . 37
- 9.4 Comparison operators . . . 39
- 9.5 Equality . . . 41

9.6 Division 42
9.7 Indexing 43
9.8 Equality tests in Scheme 45

10 * Oh My Gawd *: It's Full of Stars **49**
10.1 Surgery on nested lists 49
10.1.1 Removing a member 49
10.1.2 Inserting a member 51
10.1.3 Frequency counting 55
10.1.4 Substituting 57
10.2 `and` vs `or` 65

11 Shadows **69**
11.1 Symbols and Environments 69
11.1.1 Representations 72

12 Friends and Relations **74**
12.1 Sets and Operations 74
12.1.1 Set test 74
12.1.2 Build a set 75
12.1.3 Subsets 77
12.1.4 Equality of sets 77
12.1.5 Intersection of sets 78
12.1.6 Union of sets 78
12.1.7 Difference of sets 79
12.2 `pairs` 80
12.3 Relations 82
12.3.1 Reversing a `pair` 82
12.3.2 One to One relations 83

13 Lambda the Ultimate **85**
13.1 Higher Order Functions 85
13.1.1 Passing a Function as an Argument 85
13.1.2 Functions Returning a Function 86
13.1.3 Passing Anonymous Functions as Arguments 87
13.1.4 The Power of Functional Abstraction 90
13.1.5 Collector Functions or Continuations 93
13.1.6 Continuation Passing Style 94
13.2 Examples on Continuations 97
13.2.1 Inserting and counting: `multiinsertLR&co` 97

13.2.2 Multiple Number Operations: `evens-only*&co` 100

14 . . . and Again, and Again, and Again, . . . 104
14.1 Partial Functions 104
14.2 Recursive nature of functions 108
14.3 Turing Halting Problem 111
14.4 What is Recursion? What is a Function? 111
14.4.1 Closures and Lexical Scoping 111
14.4.2 The Y Combinator Derivation 112

15 What Is the Value of All of This? 122
15.1 Entries and Lookups 122
15.2 Values 124
15.3 Types, Value, and Meaning 125
15.4 Evaluation and Application 129

16 Intermission 133

1 What is Lisp?

1.1 History

LISP stands for List Processor or List Processing. It is a programming language which originated around 1958 at the Massachusetts Institute of Technology (MIT). The original paper titled 'Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I' was published later in 1960 by John McCarthy the creator or as some put it the discoverer of Lisp. A trivia: The term Artificial Intelligence (AI) which has eneterd the public's lexicon today was coined by John McCarthy. The orginal language for AI implementation was none other than Lisp. But the language was geared towards symbolic artifical intelliegnece. McCarthy probably wrote his paper as a theoretical exercise and one of his students Steve Russell, as legend says, implemented it overnight on an IBM 704 computer with punch cards! At one point of time there were operating systems, hardware called Lisp machines but all that is a part of computing history now.

1.2 Why Lisp?

Lisp has little to no syntax. It is very easy to pick up, I would say even easier than Python. Everything in Lisp is a singly linked list. Each element in a list is called a `cons` cell. And a `cons` is made of two components - `car` and `cdr`. `car` stands for 'contents of address register' and `cdr` stands for 'contents of decrement register'. These were catered to the IBM 704 on which it was first implemented. There is no reason to evangelize Lisp like many prominent computer scientists of the past do. My preference comes from the fact that both Lisp and Python take the reader immediately to computer science concepts without wasting time on syntax.

```
[*|*]---> cdr
|
|
V
car
```

Lisp invented a whole long list of computer science concepts which are taken for granted today. Some of which include:

- Garbage collection
- REPL

- Macros
- Homoiconicity (Code = Data)
- Dynamic Typing
- First Class Functions
- Closures
- Tail Call Optimization etc.

Lisp uses a prefix mathematical notation, meaning that we do not say `(2 + 5)` but `(+ 2 5)`. The first element could be a function, a special form, or a macro for that matter. Contents in the list get 'evaluated'. There are many dialects of Lisp, the famous ones are Common Lisp, Clojure, Racket which is a type of Scheme Lisp.

1.3 Resources to learn Lisp

The best introductory textbook in my opinion is Professor David Touretzky's book titled 'Common Lisp A Gentle Introduction to Symbolic Computing' (2nd ed.). Following that 'The Little Schemer' and 'The Seasoned Schemer'. The other option which is recommended is 'How to Design Programs: An Introduction to Programming and Computing', though I have not read it. Finally everyone needs to read 'Structure and Interpretation of Computer Programs', myself included.

A final note. The principles we learn is universal and not language dependent. So take language discussions with a pinch of salt. All the best.

2 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 at 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.

3 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. It is the same in other pursuits too such as mathematics or physics.
- 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. It is better to keep going back and working through a chapter again than to continue forward without a 100% clarity.

- 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.

4 The Five Rules

1. The Law of `car` : The primitive `car` is defined only for non-empty lists.
2. The Law of `cdr`: The primitive `cdr` is defined only for non-empty lists. The `cdr` of any non-empty list is always another list.
3. The Law of `cons`: The primitive `cons` takes two arguments. The second argument to `cons` must be a list. The result is a list.
4. The Law of `null?`: The primitive `null?` is defined only for lists.
5. The Law of `eq?`: The primitive `eq?` in takes two arguments. Each must be a non numeric atom.

5 The Ten Commandments

The First Commandment

- When recurring on a list of atoms, `lat`, ask two questions about it: `(null? lat)` and `else`. When recurring on a number, `n`, ask two questions about it: `(zero? n)` and `else`. When recurring on a list of S-expressions, `l` ask three questions about it: `(null? l)`, `(atom? (car l))`, and `else`.

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 Commandment

- Always change at least one argument while recurring. When recurring on a list of atoms, `lat`, use `(cdr lat)`. When recurring on a number, `n`, use `(sub1 n)`. And when recurring on a list of S-expressions, `l`, use `(car l)` and `(cdr l)` if neither `(null? l)` nor `(atom? (car l))` are true. 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?` 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 adding `1` does not change the value of a multiplication. When building a value with `cons`, always consider `()` for the value of the terminating line.

The Sixth Commandment

- Simplify only after the function is correct.

The Seventh Commandment

- Recur on the *subparts* that are of the same nature
  - On the sublists of a list.
  - On the sub-expression of an arithmetic expression.

The Eight Commandment

- Use help functions to abstract from representations.

The Ninth Commandment

- Abstract common patterns with a new function.

The Tenth Commandment

- Build functions to collect more than one value at a time

6 Toys

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

6.1 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
```

6.2 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 Prof 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.

6.3 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 Prof 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)
```

6.4 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
```

6.5 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

7 Do It, Do It Again, and Again, and Again ...

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

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

7.1 Basics of Recursion

- After reading Prof 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))))
```

7.1.1 The important `cond`

- 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.

7.1.2 Trace tool in Dr Racket

- 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 Prof 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.

8 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?`****.*

8.1 List Surgery

8.1.1 Removing elements

- 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)
```

8.1.2 Finding elements

- 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)`.

8.1.3 Inserting and substituting elements

- 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)
```

8.1.4 Multiple surgeries on lists

- 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.

9 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.*

9.1 Addition and Subtraction

- 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
```

9.2 Tuples

- 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 Pof 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
```

9.3 Multiplication

- 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
```

9.3.1 Digression to Abstract Algebra

- 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)
```

9.4 Comparison operators

- 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
```

9.5 Equality

- 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^
      (lambda (n m)
        (cond ((zero? m) 1)
              (else (x n (o^ 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
```

9.6 Division

- 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
```

9.7 Indexing

- 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))))))
```

9.8 Equality tests in Scheme

- In Prof 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

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

This chapter we start looking at nested lists.

Key Takeaway:
The First Commandment (final version): When recurring on a list of atoms, `lat` *ask two questions about it* `(null? lat)` *and* `else`*. When recurring on a number,* `n` *ask two questions about it:* `(zero? n)` *and* `else`*. When recurring on a list of S-expressions,* `l`*, ask three questions about it:* `(null? l)`*,* `(atom? (car l))`*, and* `else`*.*

The Fourth Commandment (final version): Always change at least one argument while recurring. When recurring on a list of atoms, `lat`*, use* `(cdr lat)`*. When recurring on a number,* `n`*, use* `(sub1 n)`*. And when recurring on a list of S-expressions,* `l`*, use* `(car l)` *and* `(cdr l)` *if neither* `(null? l)` *nor* `(atom? (car l))` *are true. 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?` *and when using* `sub1`*, test termination with* `zero?`*.*

The Sixth Commandment: Simplify only after the function is correct.

10.1 Surgery on nested lists

10.1.1 Removing a member

- 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 on the entire list. We write `rember*` which removes a specific element from even nested lists. The logic is to simply check if 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)))
```

10.1.2 Inserting a member

- `insertR*` is also easy to write but the `trace` for the example shows detailed stack calls for recursion and the true depth can be gauged as to what is happening in the machine.

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

```
> (insertR* 'roast 'chuck
            '((how much (wood))
              could
              ((a (wood) chuck))
              (((chuck)))
              (if (a) ((wood chuck)))
              could chuck wood))
>(insertR*
  'roast
  'chuck
  '((how much (wood))
    could
    ((a (wood) chuck))
    (((chuck)))
    (if (a) ((wood chuck)))
    could
    chuck
    wood))
```

```
> (insertR* 'roast 'chuck '(how much (wood)))
> >(insertR* 'roast 'chuck '(much (wood)))
> > (insertR* 'roast 'chuck '((wood)))
> > >(insertR* 'roast 'chuck '(wood))
> > > (insertR* 'roast 'chuck '())
< < < '()
< < <'(wood)
> > >(insertR* 'roast 'chuck '())
< < <'()
< < '((wood))
< <'(much (wood))
< '(how much (wood))
> (insertR*
   'roast
   'chuck
   '(could
     ((a (wood) chuck))
     (((chuck)))
     (if (a) ((wood chuck)))
     could
     chuck
     wood))
> >(insertR*
    'roast
    'chuck
    '(((a (wood) chuck)) (((chuck))) (if (a) ((wood chuck))) could chuck wood))
> > (insertR* 'roast 'chuck '((a (wood) chuck)))
> > >(insertR* 'roast 'chuck '(a (wood) chuck))
> > > (insertR* 'roast 'chuck '((wood) chuck))
> > > >(insertR* 'roast 'chuck '(wood))
> > > > (insertR* 'roast 'chuck '())
< < < < '()
< < < <'(wood)
> > > >(insertR* 'roast 'chuck '(chuck))
> > > > (insertR* 'roast 'chuck '())
< < < < '()
< < < <'(chuck roast)
< < < '((wood) chuck roast)
< < <'(a (wood) chuck roast)
> > >(insertR* 'roast 'chuck '())
```

```
< < <’()
< < ’((a (wood) chuck roast))
> > (insertR*
     ’roast
     ’chuck
     ’((((chuck))) (if (a) ((wood chuck))) could chuck wood))
> > >(insertR* ’roast ’chuck ’(((chuck))))
> > > (insertR* ’roast ’chuck ’((chuck)))
> > > >(insertR* ’roast ’chuck ’(chuck))
> > > > (insertR* ’roast ’chuck ’())
< < < < ’()
< < < <’(chuck roast)
> > > >(insertR* ’roast ’chuck ’())
< < < <’()
< < < ’((chuck roast))
> > > (insertR* ’roast ’chuck ’())
< < < ’()
< < <’(((chuck roast)))
> > >(insertR* ’roast ’chuck ’((if (a) ((wood chuck))) could chuck wood))
> > > (insertR* ’roast ’chuck ’(if (a) ((wood chuck))))
> > > >(insertR* ’roast ’chuck ’((a) ((wood chuck))))
> > > > (insertR* ’roast ’chuck ’(a))
> > > > >(insertR* ’roast ’chuck ’())
< < < < <’()
< < < < ’(a)
> > > > (insertR* ’roast ’chuck ’(((wood chuck))))
> > > > >(insertR* ’roast ’chuck ’((wood chuck)))
> > > > > (insertR* ’roast ’chuck ’(wood chuck))
> > > >[10] (insertR* ’roast ’chuck ’(chuck))
> > > >[11] (insertR* ’roast ’chuck ’())
< < < <[11] ’()
< < < <[10] ’(chuck roast)
< < < < < ’(wood chuck roast)
> > > > > (insertR* ’roast ’chuck ’())
< < < < < ’()
< < < < <’((wood chuck roast))
> > > > >(insertR* ’roast ’chuck ’())
< < < < <’()
< < < < ’(((wood chuck roast)))
< < < <’((a) ((wood chuck roast)))
```

```
< < < '(if (a) ((wood chuck roast)))
> > > (insertR* 'roast 'chuck '(could chuck wood))
> > > >(insertR* 'roast 'chuck '(chuck wood))
> > > > (insertR* 'roast 'chuck '(wood))
> > > > >(insertR* 'roast 'chuck '())
< < < < <'()
< < < < '(wood)
< < < <'(chuck roast wood)
< < < '(could chuck roast wood)
< < <'((if (a) ((wood chuck roast))) could chuck roast wood)
< < '((((chuck roast))) (if (a) ((wood chuck roast))) could chuck roast wood)
< <'(((a (wood) chuck roast))
     (((chuck roast)))
     (if (a) ((wood chuck roast)))
     could
     chuck
     roast
     wood)
< '(could
    ((a (wood) chuck roast))
    (((chuck roast)))
    (if (a) ((wood chuck roast)))
    could
    chuck
    roast
    wood)
<'((how much (wood))
   could
   ((a (wood) chuck roast))
   (((chuck roast)))
   (if (a) ((wood chuck roast)))
   could
   chuck
   roast
   wood)
'((how much (wood)) could ((a (wood) chuck roast)) (((chuck roast)))
  (if (a) ((wood chuck roast))) could chuck roast wood)
```

- All * functions work on empty lists, at atom consed onto a list, or a

list consed onto a list. * functions' additional activity is to recur into the car if required.

10.1.3 Frequency counting

- Next we do a frequency counter across a nested list and write a function called occur*.

```
(define occur*
  (lambda (a l)
  (cond ((null? l) 0)
        ((atom? (car l))
         (cond ((eq? a (car l)) (add1 (occur* a (cdr l))))
               (else (occur* a (cdr l)))))
        (else (o+ (occur* a (car l)) (occur* a (cdr l)))))))

> (occur* 'banana '((banana)
                    (split ((((banana ice)))
                            (cream (banana))
                            sherbet))
                    (banana)
                    (bread)
                    (banana brandy)))
>(occur*
  'banana
  '((banana)
    (split ((((banana ice))) (cream (banana)) sherbet))
    (banana)
    (bread)
    (banana brandy)))
> (occur* 'banana '(banana))
> >(occur* 'banana '())
< <0
< 1
> (occur*
   'banana
   '((split ((((banana ice))) (cream (banana)) sherbet))
     (banana)
     (bread)
     (banana brandy)))
```

```
> >(occur* 'banana '(split ((((banana ice))) (cream (banana)) sherbet)))
> >(occur* 'banana '(((((banana ice))) (cream (banana)) sherbet)))
> > (occur* 'banana '((((banana ice))) (cream (banana)) sherbet))
> > >(occur* 'banana '(((banana ice))))
> > > (occur* 'banana '((banana ice)))
> > > >(occur* 'banana '(banana ice))
> > > > (occur* 'banana '(ice))
> > > > (occur* 'banana '())
< < < < 0
< < < <1
> > > >(occur* 'banana '())
< < < <0
< < < 1
> > > (occur* 'banana '())
< < < 0
< < <1
> > >(occur* 'banana '((cream (banana)) sherbet))
> > > (occur* 'banana '(cream (banana)))
> > > (occur* 'banana '((banana)))
> > > >(occur* 'banana '(banana))
> > > > (occur* 'banana '())
< < < < 0
< < < <1
> > > >(occur* 'banana '())
< < < <0
< < < 1
> > > (occur* 'banana '(sherbet))
> > > (occur* 'banana '())
< < < 0
< < <1
< < 2
> > (occur* 'banana '())
< < 0
< <2
> >(occur* 'banana '((banana) (bread) (banana brandy)))
> > (occur* 'banana '(banana))
> > >(occur* 'banana '())
< < <0
< < 1
> > (occur* 'banana '((bread) (banana brandy)))
```

```
> > >(occur* 'banana '(bread))
> > >(occur* 'banana '())
< < <0
> > >(occur* 'banana '((banana brandy)))
> > > (occur* 'banana '(banana brandy))
> > > >(occur* 'banana '(brandy))
> > > >(occur* 'banana '())
< < < <0
< < < 1
> > > (occur* 'banana '())
< < < 0
< < <1
< < 1
< <2
< 4
<5
5
```

10.1.4 Substituting

- Similarly we write the `*` function for `subst*`. Nothing additional other than keeping in mind that we have to recur inside `car` of `l` also.

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

```
> (subst* 'orange 'banana
          '((banana)
            (split ((((banana ice)))
                    (cream (banana))
                    sherbet))
```

```
            (banana)
            (bread)
            (banana brandy)))
>(subst*
  'orange
  'banana
  '((banana)
    (split ((((banana ice))) (cream (banana)) sherbet))
    (banana)
    (bread)
    (banana brandy)))
> (subst* 'orange 'banana '(banana))
> >(subst* 'orange 'banana '())
< <'()
< '(orange)
> (subst*
   'orange
   'banana
   '((split ((((banana ice))) (cream (banana)) sherbet))
     (banana)
     (bread)
     (banana brandy)))
> >(subst*
    'orange
    'banana
    '(split ((((banana ice))) (cream (banana)) sherbet)))
> > (subst* 'orange 'banana '(((((banana ice))) (cream (banana)) sherbet)))
> > >(subst* 'orange 'banana '((((banana ice))) (cream (banana)) sherbet))
> > > (subst* 'orange 'banana '(((banana ice))))
> > > >(subst* 'orange 'banana '((banana ice)))
> > > > (subst* 'orange 'banana '(banana ice))
> > > > >(subst* 'orange 'banana '(ice))
> > > > > (subst* 'orange 'banana '())
< < < < < '()
< < < < <'(ice)
< < < < '(orange ice)
> > > > (subst* 'orange 'banana '())
< < < < '()
< < < <'((orange ice))
> > > >(subst* 'orange 'banana '())
```

```
< < < <'()
< < < '(((orange ice)))
> > > (subst* 'orange 'banana '((cream (banana)) sherbet))
> > > >(subst* 'orange 'banana '(cream (banana)))
> > > > (subst* 'orange 'banana '((banana)))
> > > > >(subst* 'orange 'banana '(banana))
> > > > > (subst* 'orange 'banana '())
< < < < < '()
< < < < <'(orange)
> > > > >(subst* 'orange 'banana '())
< < < < <'()
< < < < '((orange))
< < < <'(cream (orange))
> > > >(subst* 'orange 'banana '(sherbet))
> > > > (subst* 'orange 'banana '())
< < < < '()
< < < <'(sherbet)
< < < '((cream (orange)) sherbet)
< < <'((((orange ice))) (cream (orange)) sherbet)
> > >(subst* 'orange 'banana '())
< < <'()
< < '(((((orange ice))) (cream (orange)) sherbet))
< <'(split ((((orange ice))) (cream (orange)) sherbet))
> >(subst* 'orange 'banana '((banana) (bread) (banana brandy)))
> > (subst* 'orange 'banana '(banana))
> > >(subst* 'orange 'banana '())
< < <'()
< < '(orange)
> > (subst* 'orange 'banana '((bread) (banana brandy)))
> > >(subst* 'orange 'banana '(bread))
> > > (subst* 'orange 'banana '())
< < < '()
< < <'(bread)
> > >(subst* 'orange 'banana '((banana brandy)))
> > > (subst* 'orange 'banana '(banana brandy))
> > > >(subst* 'orange 'banana '(brandy))
> > > > (subst* 'orange 'banana '())
< < < < '()
< < < <'(brandy)
< < < '(orange brandy)
```

```
> > > (subst* 'orange 'banana '())
< < < '()
< < <'((orange brandy))
< < '((bread) (orange brandy))
< <'((orange) (bread) (orange brandy))
< '((split ((((orange ice))) (cream (orange)) sherbet))
    (orange)
    (bread)
    (orange brandy))
<'((orange)
   (split ((((orange ice))) (cream (orange)) sherbet))
   (orange)
   (bread)
   (orange brandy))
'((orange)
  (split
   ((((orange ice)))
    (cream (orange))
    sherbet))
  (orange)
  (bread)
  (orange brandy))
```

- The `insertL*` is again on the same lines as before. Planning to omit trace flows now since it will take multiple pages.

```
(define insertL*
  (lambda (new old l)
    (cond ((null? l) (quote ()))
          ((atom? (car l))
           (cond ((eq? (car l) old)
                  (cons new (cons old (insertL* new old (cdr l)))))
                 (else (cons (car l) (insertL* new old (cdr l))))))
          (else (cons (insertL* new old (car l))
                      (insertL* new old (cdr l)))))))
```

```
> (insertL* 'pecker 'chuck
            '((how much (wood))
              could
```

```
              ((a (wood) chuck))
              (((chuck)))
              (if (a) ((would chuck)))
              could chuck wood))
>(insertL*
  'pecker
  'chuck
  '((how much (wood))
    could
    ((a (wood) chuck))
    (((chuck)))
    (if (a) ((would chuck)))
    could
    chuck
    wood))
> (insertL* 'pecker 'chuck '(how much (wood)))
> >(insertL* 'pecker 'chuck '(much (wood)))
> > (insertL* 'pecker 'chuck '((wood)))
> > >(insertL* 'pecker 'chuck '(wood))
> > > (insertL* 'pecker 'chuck '())
< < < '()
< < <'(wood)
> > >(insertL* 'pecker 'chuck '())
< < <'()
< < '((wood))
< <'(much (wood))
< '(how much (wood))
> (insertL*
   'pecker
   'chuck
   '(could
     ((a (wood) chuck))
     (((chuck)))
     (if (a) ((would chuck)))
     could
     chuck
     wood))
> >(insertL*
    'pecker
    'chuck
```

```
    '(((a (wood) chuck))
      (((chuck)))
      (if (a) ((would chuck)))
      could
      chuck
      wood))
> > (insertL* 'pecker 'chuck '((a (wood) chuck)))
> > >(insertL* 'pecker 'chuck '(a (wood) chuck))
> > > (insertL* 'pecker 'chuck '((wood) chuck))
> > > >(insertL* 'pecker 'chuck '(wood))
> > > > (insertL* 'pecker 'chuck '())
< < < < '()
< < < <'(wood)
> > > >(insertL* 'pecker 'chuck '(chuck))
> > > > (insertL* 'pecker 'chuck '())
< < < < '()
< < < <'(pecker chuck)
< < < '((wood) pecker chuck)
< < <'(a (wood) pecker chuck)
> > >(insertL* 'pecker 'chuck '())
< < <'()
< < '((a (wood) pecker chuck))
> > (insertL*
     'pecker
     'chuck
     '((((chuck))) (if (a) ((would chuck))) could chuck wood))
> > >(insertL* 'pecker 'chuck '(((chuck))))
> > > (insertL* 'pecker 'chuck '((chuck)))
> > > >(insertL* 'pecker 'chuck '(chuck))
> > > > (insertL* 'pecker 'chuck '())
< < < < '()
< < < <'(pecker chuck)
> > > >(insertL* 'pecker 'chuck '())
< < < <'()
< < < '((pecker chuck))
> > > (insertL* 'pecker 'chuck '())
< < < '()
< < <'(((pecker chuck)))
> > >(insertL* 'pecker 'chuck '((if (a) ((would chuck))) could chuck wood))
> > > (insertL* 'pecker 'chuck '(if (a) ((would chuck))))
```

```
> > > >(insertL* 'pecker 'chuck '((a) ((would chuck))))
> > > > (insertL* 'pecker 'chuck '(a))
> > > > >(insertL* 'pecker 'chuck '())
< < < < <'()
< < < < '(a)
> > > > (insertL* 'pecker 'chuck '(((would chuck))))
> > > > >(insertL* 'pecker 'chuck '((would chuck)))
> > > > > (insertL* 'pecker 'chuck '(would chuck))
> > > >[10] (insertL* 'pecker 'chuck '(chuck))
> > > >[11] (insertL* 'pecker 'chuck '())
< < < <[11] '()
< < < <[10] '(pecker chuck)
< < < < < '(would pecker chuck)
> > > > > (insertL* 'pecker 'chuck '())
< < < < < '()
< < < < <'((would pecker chuck))
> > > > >(insertL* 'pecker 'chuck '())
< < < < <'()
< < < < '(((would pecker chuck)))
< < < <'((a) ((would pecker chuck)))
< < < '(if (a) ((would pecker chuck)))
> > > (insertL* 'pecker 'chuck '(could chuck wood))
> > > >(insertL* 'pecker 'chuck '(chuck wood))
> > > > (insertL* 'pecker 'chuck '(wood))
> > > > >(insertL* 'pecker 'chuck '())
< < < < <'()
< < < < '(wood)
< < < <'(pecker chuck wood)
< < < '(could pecker chuck wood)
< < <'((if (a) ((would pecker chuck))) could pecker chuck wood)
< < '((((pecker chuck)))
      (if (a) ((would pecker chuck)))
      could
      pecker
      chuck
      wood)
< <'(((a (wood) pecker chuck))
     (((pecker chuck)))
     (if (a) ((would pecker chuck)))
     could
```

```
     pecker
     chuck
     wood)
< '(could
    ((a (wood) pecker chuck))
    (((pecker chuck)))
    (if (a) ((would pecker chuck)))
    could
    pecker
    chuck
    wood)
<'((how much (wood))
   could
   ((a (wood) pecker chuck))
   (((pecker chuck)))
   (if (a) ((would pecker chuck)))
   could
   pecker
   chuck
   wood)
'((how much (wood))
  could
  ((a (wood) pecker chuck))
  (((pecker chuck)))
  (if (a) ((would pecker chuck)))
  could
  pecker
  chuck
  wood)
```

- Writing a `member*` is also fairly simple. I think so far recursion has been ingrained into this book solvers mind! The trace diagram shows which `chips` the function found. That is important.

```
(define member*
 (lambda (a l)
   (cond ((null? l) #f)
         ((atom? (car l))
          (or (eq? (car l) a)
```

```
                (member* a (cdr l))))
           (else (or (member* a (car l))
                     (member* a (cdr l)))))))
```

```
> (member* 'chips '((potato) (chips ((with) fish) (chips))))
>(member* 'chips '((potato) (chips ((with) fish) (chips))))
> (member* 'chips '(potato))
> (member* 'chips '())
< #f
>(member* 'chips '((chips ((with) fish) (chips))))
> (member* 'chips '(chips ((with) fish) (chips)))
< #t
<#t
#t
```

- `leftmost` recurs only on the `car` looking for the atom in a non-empty list.

```
(define leftmost
  (lambda (l)
    (cond ((atom? (car l)) (car l))
          (else (leftmost (car l))))))
```

```
> (leftmost '((potato) (chips ((with) fish) (chips))))
>(leftmost '((potato) (chips ((with) fish) (chips))))
>(leftmost '(potato))
<'potato
'potato
```

10.2 `and` vs `or`

- Again the distinction between `and` and `or` is made. For `and` if the any of the expressions evaluates to `#f` then further evaluation stops. In `or` the evaluation stops the moment we find a `#t`.
- In a footnote a very important property of `cond` is highlighted (mentioned earlier in this booklet also). `cond` does not consider all of its arguments. It essentially goes through a sequence from top to bottom. Prof Touretzky's book has a crystal clear explanation on this. In

this book the authors state that neither `and` nor `or` can be stated as functions using `cond` though both can be expressed as abbreviations of `cond`.

```
(and a b) = (cond (a b) else #f)
(or a b) = (cond (a #t) (else b))
```

- The function `eqlist?` is written to compare the elements of two lists. The first attempt is rather elaborate building from first logical principles. But the second is concise. Here I write the second version.

```
(define eqlist?
  (lambda (l1 l2)
    (cond ((and (null? l1) (null? l2)) #t)
          ((or (null? l1) (null? l2)) #f)
          ((and (atom? (car l1)) (atom? (car l2)))
           (and (eqan? (car l1) (car l2))
                (eqlist? (cdr l1) (cdr l2))))
          ((or (atom? (car l1)) (atom? (car l2))) #f)
          (else (eqlist? (car l1) (car l2))
                (eqlist? (cdr l1) (cdr l2))))))
```

```
> (eqlist? '(aloo ((matar)) (and (gobhi)))
           '(aloo ((matar)) and (gobhi)))
>(eqlist? '(aloo ((matar)) (and (gobhi))) '(aloo ((matar)) and (gobhi)))
>(eqlist? '(((matar)) (and (gobhi))) '(((matar)) and (gobhi)))
> (eqlist? '((matar)) '((matar)))
> >(eqlist? '(matar) '(matar))
> >(eqlist? '() '())
< <#t
> (eqlist? '() '())
< #t
>(eqlist? '((and (gobhi))) '(and (gobhi)))
<#f
#f

> (eqlist? '(aloo ((matar)) (and (gobhi)))
           '(aloo ((gajar)) and (gobhi)))
>(eqlist? '(aloo ((matar)) (and (gobhi))) '(aloo ((gajar)) and (gobhi)))
>(eqlist? '(((matar)) (and (gobhi))) '(((gajar)) and (gobhi)))
```

```
> (eqlist? '((matar)) '((gajar)))
> >(eqlist? '(matar) '(gajar))
< <#f
> (eqlist? '() '())
< #t
>(eqlist? '((and (gobhi))) '(and (gobhi)))
<#f
#f
```

- The book reiterates the definition of S-expression again: It is an atom or a (possibly empty) list of S-expressions.
- Here I think is the turning point of the book. I think the basics are behind us now.
- The function `equal?` is written using `eqlist?`. I am calling it `my-equal?` since `equal?` is built in to the language. The catch is now `my-equal?` is comparing two S-expressions and not just atoms.

```
(define my-equal?
  (lambda (s1 s2)
    (cond ((and (atom? s1) (atom? s2)) (eqan? s1 s2))
          ((or (atom? s1) (atom? s2)) #f)
          (else (eqlist? s1 s2)))))

(define eqlist-equal?
  (lambda (l1 l2)
    (cond ((and (null? l1) (null? l2)) #t)
          ((or (null? l1) (null? l2)) #f)
          (else (and (my-equal? (car l1) (car l2)))
                (eqlist-equal? (cdr l1) (cdr l2))))))
```

- `rember` is written so that it operates on an S-expression. The original function on a list of atoms was elegant itself. But the good part is we are now operating on S-expressions rather than atoms. Anyways here is the function

```
(define rember-equal
  (lambda (s l)
```

```
(cond
  ((null? l) (quote ()))
  ((equal? (car l) s) (cdr l))
  (else (cons (car l)
              (rember-equal s (cdr l)))))))
```

- We cannot simplify `insertL*` because this function actually needs to look at atoms specifically.
- The last question in the chapter is important. We should know which comparison predicate we need to use when. The table which I have in the previous chapter is good for that.

Core Terms/Concepts Learnt

- Able to recur in the `car`.
- All comparison operators in the language (sort of) `eq?`, `=`, `equal` primarily.
- Recurring over a list of atoms vs S-expressions.

11 Shadows

Names can hide other names. Environments matter.

Key Takeaway:
The Seventh Commandment: Recur on the sub-parts that are of the same nature: 1) On the sub-lists of a list, 2) On the sub-expressions of an arithmetic expression.

The Eight Commandment: Use help functions to abstract from representations.

11.1 Symbols and Environments

- We enter the realm of what symbols actually mean. A `+` may mean just a symbol and not an operation of addition. So we may have a traditional arithmetic operation of `2 + 4 x 5` which is an arithmetic expression. But this expression `2 + 4 x 5` can be bound to a symbol `cookie`. Thus `cookie` is also an arithmetic expression.

  ```
  (define cookie 12)
  ```

- The context or the environment matters.

- A traditional `(n + 5)` is not considered an arithmetic expression since the parenthesis is not included in the definition of arithmetic expressions.

- But `(n + 5)` is an S-expression. Therefore we can use it as an argument to a function. So how do we represent `3 + 4 x 5`? By `(3 + ( 4 x 5))`.

- `numbered?` is a function that determines if a representation of an arithmetic expressions contains only numbers apart from +, x, exponential (`o^`). It is not actually evaluating the arithmetic expression.

- The function `numbered?` is basically checking for the structural object of an arithmetic expression. I believe this is the early part of an interpreter.

- The authors explicitly call out that `(1 + 4)` is not a list but in a list form.

- I have not yet come across the reason why subtraction or division are not included in this basic interpretation of arithmetic expressions. Perhaps going into negative integers and for division going into the realm of decimals (floating points).

- The first question asked in `numbered?` is whether all arithmetic expressions that are atoms are numbers or not. But why could not we only ask if it is a number only?

- Now we can take a first stab at writing the `numbered?` function

```
(define numbered-long?
  (lambda (aexp)
    (cond ((atom? aexp) (number? aexp))
          ((eq? (car (cdr aexp)) (quote o+))
           (and (numbered-long? (car aexp))
                (numbered-long? (car (cdr (cdr aexp))))))
          ((eq? (car (cdr aexp)) (quote o*))
           (and (numbered-long? (car aexp))
                (numbered-long? (car (cdr (cdr aexp))))))
          ((eq? (car (cdr aexp)) (quote o^))
           (and (numbered-long? (car aexp))
                (numbered-long? (car (cdr (cdr aexp)))))))))

(define numbered?
  (lambda (aexp)
    (cond ((atom? aexp) (number? aexp))
          (else (and (numbered? (car aexp))
                     (numbered? (car (cdr (cdr aexp))))))))))
```

- `value` is a function which returns the natural value of a numbered arithmetic expression. In the book the operations are limited to addition, multiplication and exponents. No error handling or any other arithmetic operation. It is fine since the intent is to get conceptual clarity and not go for a full blown interpreter.

```
(define value
  (lambda (nexp)
    (cond ((atom? nexp) nexp)
          ((eq? (car (cdr nexp)) (quote +))
```

```
                (o+ (value (car nexp))
                    (value (car (cdr (cdr nexp))))))
               ((eq? (car (cdr nexp)) (quote x))
                (o* (value (car nexp))
                    (value (car (cdr (cdr nexp))))))
               (else
                (o^ (value (car nexp))
                    (value (car (cdr (cdr nexp)))))))))
```

```
> (value '((2 + 5) x (4 x 2)))
>(value '((2 + 5) x (4 x 2)))
> (value '(2 + 5))
> >(value 2)
< <2
> >(value 5)
< <5
< 7
> (value '(4 x 2))
> >(value 4)
< <4
> >(value 2)
< <2
< 8
<56
56
```

- A wrong version of `value` for a prefix notation syntax is written and the concept of helper functions is introduced. The helper functions will pick the sub expressions. This is so because in the incorrect value function for prefix notation is wrong because it ends up with a subpart which is not an arithmetic expression. Even the operator can be got with a helper function easily. For the normal notation also these helper functions can be changed (earlier code on this).

```
(define 1st-sub-exp
  (lambda (aexp)
    (car (cdr aexp))))

(define 2nd-sub-exp
```

```
  (lambda (aexp)
    (car (cdr (cdr aexp)))))

(define operator
  (lambda (aexp)
    (car aexp)))
```

- Finally writing the function `value` with help of helper functions.

```
    (define value-help
      (lambda (nexp)
        (cond ((atom? nexp) nexp)
              ((eq? (operator nexp) (quote +))
               (o+ (value-help (1st-sub-exp nexp))
                   (value-help (2nd-sub-exp nexp))))
              ((eq? (operator nexp) (quote x))
               (o* (value-help (1st-sub-exp nexp))
                   (value-help (2nd-sub-exp nexp))))
              (else
               (o^ (value (1st-sub-exp nexp))
                   (value (2nd-sub-exp nexp)))))))

> (value-help '(+ 24 25))
49
```

11.1.1 Representations

- The idea of representations is explicitly stated. `4` can be stated as say for instance `(() () () ())` or even `(I V)`. The book extends the same representations for a few natural numbers. It is self explanatory.

- We write a function called `sero?` for the representation of numbers in terms of `()` (same example as above). We also write `edd1` and `zub1`. The authors have given such an elegant example of representation here.

```
(define sero?
  (lambda (n)
    (null? n)))
```

```
(define edd1
  (lambda (n)
    (cons (quote ()) n)))

(define zub1
  (lambda (n)
    (cdr n)))
```

- A `(zub1 n)` on `()` as `n` will give no answer as per the book. Theoretically speaking (negative numbers we would not be able to reproduce).
- Let us write `o+rep`.

```
(define o+rep
  (lambda n m
    (cond ((sero? m) n)
          (else (edd1 (o+rep n (zub1 m)))))))
```

- The last two questions for the checking whether the new representation of `4` in parenthesis terms is a `lat?` or not. We must be aware of shadows because new binding of a variable hides the old binding.The interpreter looks for the nearest binding first.

```
> (lat? '((()) (()()) (()()()())))
#f> (lat? '((()) (()()) (()()()())))
#f
```

Core Terms/Concepts Learnt

- Names are not their values, the environment determines the meaning. Meaning of symbol depends on the current context/environment.
- Expressions can represent numbers. Representation and value of an arithmetic expression is showing what it is and what value it evaluates to.
- Recurring over a list of atoms vs S-expressions.
- Rebinding names hides or shadows previous meanings. *You must be aware of shadows.*

12 Friends and Relations

Working with relations and symbolic computation. Manipulate lists of symbols and represent structured data.

Key Takeaway:
Sets and set operations: Building and testing them.

Relations, Pairs and One-to-One Functions: Defining them.

12.1 Sets and Operations

- `set?` is exactly like the mathematical set in set theory. The book at the end also recommends Naive Set Theory as an interesting read by the great Paul Halmos. I own a copy but I am yet to read. The other book I like is Book of Proof by Richard Hammack.
- A set has unique elements. No element in the set repeats.
- An empty list `()` has no atoms. Thus nothing repeats and therefore this is a set too.

12.1.1 Set test

- Writing `set?` is fairly easy

```
(define set?
  (lambda (lat)
    (cond ((null? lat) #t)
          ((member? (car lat) (cdr lat)) #f)
          (else (set? (cdr lat))))))
```

```
> (set? '(pani puri gol guppa phucka pani))
>(set? '(pani puri gol guppa phucka pani))
<#f
#f
> (set? '(pani puri gol guppa phucka))
>(set? '(pani puri gol guppa phucka))
>(set? '(puri gol guppa phucka))
>(set? '(gol guppa phucka))
>(set? '(guppa phucka))
```

```
>(set? '(phucka))
>(set? '())
<#t
#t
```

12.1.2 Build a set

- Next we build sets. First we write `makeset` using `member?` and then with `multirember`. There is a difference between the two methods. In using `member?` we go through each atom on the list one by one and only consing that when it is not in the rest of the list. It is building using a predicate test actually. While using `multirember` we first build a list and that list is build by removing the elements given to the `makeset` function. Trace should show the flow cleanly.

```
(define makeset
  (lambda (lat)
    (cond ((null? lat) (quote ()))
          ((member? (car lat) (cdr lat))
           (makeset (cdr lat)))
          (else (cons (car lat) (makeset (cdr lat)))))))
```

```
> (makeset '(apple peach pear peach
                  plum apple lemon peach))
>(makeset '(apple peach pear peach plum apple lemon peach))
>(makeset '(peach pear peach plum apple lemon peach))
>(makeset '(pear peach plum apple lemon peach))
> (makeset '(peach plum apple lemon peach))
> (makeset '(plum apple lemon peach))
> >(makeset '(apple lemon peach))
> > (makeset '(lemon peach))
> > >(makeset '(peach))
> > > (makeset '())
< < < '()
< < <'(peach)
< < '(lemon peach)
< <'(apple lemon peach)
< '(plum apple lemon peach)
<'(pear plum apple lemon peach)
```

```
'(pear plum apple lemon peach)
```

```
    (define makeset-multirember
      (lambda (lat)
        (cond ((null? lat) (quote ()))
              (else
               (cons (car lat)
                     (makeset-multirember
                      (multirember (car lat) (cdr lat))))))))
```

```
> (makeset-multirember '(apple peach pear peach
                                plum apple lemon peach))
>(makeset-multirember '(apple peach pear peach plum apple lemon peach))
> (makeset-multirember '(peach pear peach plum lemon peach))
> >(makeset-multirember '(pear plum lemon))
> > (makeset-multirember '(plum lemon))
> > >(makeset-multirember '(lemon))
> > > (makeset-multirember '())
< < < '()
< < <'(lemon)
< < '(plum lemon)
< <'(pear plum lemon)
< '(peach pear plum lemon)
<'(apple peach pear plum lemon)
'(apple peach pear plum lemon)
```

- Additionally testing `makeset` on data which has numbers.

```
> (makeset '(apple 3 pear 4 9 apple 3 4))
'(pear 9 apple 3 4)
```

```
> (makeset-multirember '(apple 3 pear 4 9 apple 3 4))
>(makeset-multirember '(apple 3 pear 4 9 apple 3 4))
> (makeset-multirember '(3 pear 4 9 3 4))
> >(makeset-multirember '(pear 4 9 4))
> > (makeset-multirember '(4 9 4))
> > >(makeset-multirember '(9))
> > > (makeset-multirember '())
```

```
< < < '()
< < <'(9)
< < '(4 9)
< <'(pear 4 9)
< '(3 pear 4 9)
<'(apple 3 pear 4 9)
'(apple 3 pear 4 9)
```

12.1.3 Subsets

- `subset?` is a test to check if one set is a part of the other set. Each element of the first set should be present in the other set for it to be called a subset. This is elementary set theory.

```
(define subset?
  (lambda (set1 set2)
    (cond ((null? set1) #t)
          ((member? (car set1) set2)
           (subset? (cdr set1) set2))
          (else #f))))
```

The book asks us to write `subset?` with an `and` also. At first we might think why but I am guessing the authors will drive home the point at a later point in time.

```
(define subset-and?
  (lambda (set1 set2)
    (cond ((null? set1) #t)
          (else (and (member? (car set1) set2)
                     (subset-and? (cdr set1) set2))))))
```

12.1.4 Equality of sets

- Next is the test to see if the sets are exactly the same. We can actually just check is the first set is the subset of the second one and the second set is also the subset of the first one. If both are true then we can say they are equal sets. Also we truly just need one `and` test that is all. No `cond` or anything in this case. I guess this is like anonymous functions

perhaps. This is probably the first function we have written without `cond` so far.

```
(define eqset?
  (lambda (set1 set2)
    (and (subset? set1 set2)
         (subset? set2 set1))))
```

12.1.5 Intersection of sets

- The predicate `intersect?` simply checks if the any of the elements in the first set is present in the other set or not. The book asks us to simplify and also use `or` but to me without the `or` looks more clean and elegant. With `or` we simply push the last two tests of the cond with an `or` in the else leg.

```
(define intersect?
  (lambda (set1 set2)
    (cond ((null? set1) #f)
          ((member? (car set1) set2) #t)
          (else (intersect? (cdr set1) set2)))))
```

- `intersect` basically returns the elements which are common in the two sets.

```
(define intersect
  (lambda (set1 set2)
    (cond ((null? set1) (quote ()))
          ((member? (car set1) set2)
           (cons (car set1) (intersect (cdr set1) set2)))
          (else (intersect (cdr set1) set2)))))
```

12.1.6 Union of sets

- `union` picks all the elements across the two sets without repeating the intersect elements. It is exactly the same as union in set theory.

```
(define union
  (lambda (set1 set2)
    (cond ((null? set1) set2)
          ((member? (car set1) set2)
           (union (cdr set1) set2))
          (else (cons (car set1)
                      (union (cdr set1) set2))))))

> (union '(a b) '(a c d))
>(union '(a b) '(a c d))
>(union '(b) '(a c d))
> (union '() '(a c d))
< '(a c d)
<'(b a c d)
'(b a c d)
```

12.1.7 Difference of sets

- This function is the difference in the two sets. Compared to `union` the only change is that when the first set is null then we return an empty list rather than the second list.

```
(define set-difference
  (lambda (set1 set2)
    (cond ((null? set1) (quote ()))
          ((member? (car set1) set2)
           (set-difference (cdr set1) set2))
          (else (cons (car set1)
                      (set-difference (cdr set1) set2))))))
```

- This section of the chapter is interesting as we will get introduced to the concepts of pairs, relations, functions, one to one relationships. The authors say so much with so little. This is indeed a deep book and requires multiple readings. I am certain if I start reading from the beginning again I will learn even more things. Additionally, we need to keep referencing other material at the same time.

- The function `intersectall` is written very smartly! It looks at a list of lists and finds common intersection across all the lists.

```
(define intersectall ;;smart function!
  (lambda (l-set)
    (cond ((null? (cdr l-set)) (car l-set))
          (else (intersect (car l-set)
                           (intersectall (cdr l-set)))))))
```

```
> (intersectall '((a b c) (c a d e) (e f g h a b)))
>(intersectall '((a b c) (c a d e) (e f g h a b)))
> (intersectall '((c a d e) (e f g h a b)))
> >(intersectall '((e f g h a b)))
< <'(e f g h a b)
< '(a e)
<'(a)
'(a)
```

12.2 pairs

- Now we come to one of the most important things in Lisp in general - a pair. In Lisp a pair is the most fundamental data structure. It is created with the function `cons`, which stands for *construct*.

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

  The 'a' is the `car` and 'b' is `cdr`. When the `cdr` is the null list `()` then we get the list data structure. The book says a list with two atoms is a pair. That is correct but in this case the it is a special form of the pair as mentioned in the previous statement. Even a list of two S-expressions is a pair.

- The code definition of a `pair?` is quite good.

  ```
  (define a-pair?  ;;reason out each line of cond
    (lambda (x)
      (cond
        ((atom? x) #f) ;; there is not a single element
        ((null? x) #f) ;; the list is not empty
        ((null? (cdr x)) #f) ;; the list does not have only one element
        ((null? (cdr (cdr x))) #t) ;; there is no third or more element
  ```

```
                                        ;;i.e. now we have exactly 2 elements
      (else #f)))) ;; everything else is not a pair
```

- The book talks about how to build a pair exactly like I mentioned earlier. Also how to get the first and second element. In Common Lisp (Prof Touretzky's book) it was also demonstrated exactly in this manner but right at the start of the book essentially. These functions are used to make representations of pairs and to get parts of representations of pairs. They will improve readability.

```
(define my-first
  (lambda (p)
    (cond
      (else (car p)))))

(define my-second
  (lambda (p)
    (cond
      (else (car (cdr p))))))

(define my-third ;; not applicable to pairs but in general for a list
  (lambda (p)
    (cond
      (else (car (cdr (cdr p)))))))

(define build
  (lambda (s1 s2)
    (cond
      (else (cons s1 (cons s2 (quote ()))))))))
```

- We truly do not need `cond` in the definition of `first`, `second` or `third`. For instance

```
(define my-third-nocond
  (lambda (p)
    (car (cdr (cdr p)))))
```

12.3 Relations

- `rel` is a relation. Relations are between a set of pairs. This again originates from the concept of relations in set theory. A pair can be ordered or unordered. In unordered pairs `(a b)` is same as `(b a)`. But in ordered pair these two are different. A relation from one set to another is a forming of a pair where one element from the first element is taken and the second element of the pair is got from the second set. This is what we call a relationship. Now if that relation has a constraint that the first element is mapped to exactly one element in the second set then we say it is a function. So function is a one to one relationship pair between two sets.

- The book smartly defines functions as a special case of relations by saying that the first element of the pairs should be a set essentially meaning that there is one to element in first set which maps uniquely to a second element in the second set. A first element mapping to two elements in second set is not a function.

```
(define fun?
  (lambda (rel)
    (set? (firsts rel))))
```

- The next function is reversing the elements within a relation. Here we see the usefulness of `build`, `first`,and `second`. Helper functions make code look clean and flow logically.

```
(define revrel
  (lambda (rel)
    (cond ((null? rel) (quote ()))
          (else (cons (build
                       (second (car rel))
                       (first (car rel)))
                      (revrel (cdr rel)))))))
```

12.3.1 Reversing a `pair`

- Another helper function can be written which simply reverses a pair.

```
(define revpair
  (lambda (pair)
    (build (second pair) (first pair))))

(define revrel-revpair
  (lambda (rel)
    (cond ((null? rel) (quote ()))
          (else (cons (revpair (car rel))
                      (revrel-revpair (cdr rel)))))))
```

12.3.2 One to One relations

- Finally we reach the one-to-one function. In set theory this is a function (meaning that each of the element from first set i.e. the domain uniquely maps to an element in the second set i.e. the co-domain) which is one-to-one. By this we mean that each element in the second set i.e. the co-domain maps to exactly one element in the first set i.e. the domain. Therefore there is a one-to-one mapping from domain to co-domain and vice versa.

```
(define fullfun?
  (lambda (fun)
    (set? (seconds fun))))

(define one-to-one?
  (lambda (fun)
    (fun? (revrel fun))))
```

- The last page of this chapter looks like a fun thing but there is probably a lesson or two in this. First the lambda does not take any arguments so this is a procedure when called will execute a certain steps without any external influence in terms of arguments. Next we can see the sequence of multiple mixes and cream. So we actually know from the various indentations when that specific mix is done. So in a Lisp program we can communicate ideas in sequences and not be worried about instructing the computer but rather use it as a medium to communicate ideas.

Core Terms/Concepts Learnt

- Sets, Relations, Pairs, Functions, One-to-one functions
- Set operations such as intersection, union, difference, subset
- Use of helper functions

13 Lambda the Ultimate

Introduce anonymous functions and higher order design. `lambdas` encapsulate behavior and enable abstraction without naming.

Key Takeaway:
The Ninth Commandment: Abstract common patterns with a new function.

The Tenth Commandment: Build functions to collect more than one value at a time.

- This is an important chapter and I will try and make the notes as detailed as possible and have subsections for each concepts covered.

13.1 Higher Order Functions

13.1.1 Passing a Function as an Argument

- We re-look at the function to remove a member `rember` with the test being executed by `equal`. The learning is that we can create another function `rember-f` which takes in a function additionally as an argument and applies it to the `rember` function. We are learning to pass functions to functions perhaps here.

- We have 4 functions now which do similar things now `rember` with `=`, `eq?`, and `equal?`, and the one which we write now `rember-f`.

```
(define rember-f1
  (lambda (test? a l)
    (cond
      ((null? l) (quote ()))
      ((test? a (car l)) (cdr l))
      (else (cons (car l) (rember-f1 test? a (cdr l)))))))
```

- In the above version of `rember-f` we are passing a 'test function' such as `eq?` and this `eq?` gets applied within the function `rember-f`.

- When `=` passed we compare numbers, when `eq?` is passed we compare objects such as symbols or strings and when `equal?` is passed we compare structures like a list.

13.1.2 Functions Returning a Function

- So far we have seen functions returning lists and atoms.
- We get to know in this section that functions can return functions too.
- `(lambda (a l)...)` is a function which takes two arguments `a` and `l`.
- Moses Schönfinkel was a mathematician who was the inventor of combinatory logic. He was the first person to show that functions that take two or more arguments could be reduced to functions by taking only a single argument. Later the mathematician Haskell Curry after whom the programming language Haskell is named took this concept mainstream and it has been called Currying ever since. Unfortunately Schönfinkel spent his last days in poverty. After he was gone his papers were burnt down by neighbors in Moscow for heating! There are only two papers by him which remain. Everything else was lost. We do not know where those ideas could have led us to.
- Using `define` we give the lambda function a name. For instance in the case below.

```
(define eq?-c
  (lambda (a)
    (lambda (x)
      (eq? x a))))
```

- In the function above if we pass something as the argument the function returns a function object. For instance in my case Dr Racket mentions that it is returning a procedure.

```
> (eq?-c 'salad)
#<procedure:...chemer-workbook.rkt:593:4>
```

- Without giving a name to this function we can still pass the two arguments to the function.

```
> (eq?-salad 'salad)
>(eq?-salad 'salad)
<#t
```

```
#t
> (eq?-salad 'tuna)
>(eq?-salad 'tuna)
<#f
#f
> ((eq?-c 'salad) 'salad)
#t
> ((eq?-c 'salad) 'tuna)
#f
```

- Now we will curry the `rember-f` function (it is actually only partially curried). Now this function returns another function after taking only argument.

```
(define rember-f
  (lambda (test?)
    (lambda (a l)
      (cond
        ((null? l) (quote ()))
        ((test? a (car l)) (cdr l))
        (else (cons (car l)
                    ((rember-f test?) a (cdr l))))))))
> ((rember-f eq?) 'tuna '(shrimp salad and tuna salad))
'(shrimp salad and salad)
> ((rember-f eq?) 'eq? '(equal? eq? eqan? eqlist? eqpair?))
'(equal? eqan? eqlist? eqpair?)
```

13.1.3 Passing Anonymous Functions as Arguments

- The insert left or right functions can also be partially curried. One question asked is what is the difference between these two. Its only the order of `cons` in the second test that is all.

```
(define insertL-f
  (lambda (test?)
    (lambda (new old l)
      (cond
        ((null? l) (quote ()))
```

```
        ((test? (car l) old)
         (cons new (cons old (cdr l))))
        (else (cons (car l)
                    ((insertL-f test?) new old (cdr l))))))))

(define insertR-f
  (lambda (test?)
    (lambda (new old l)
      (cond
        ((null? l) (quote ()))
        ((test? (car l) old)
         (cons old (cons new (cdr l))))
        (else (cons (car l)
                    ((insertR-f test?) new old (cdr l))))))))
```

- The difference in the two can written separately as helper functions and then we can call a generic `insert-g` with these helper functions to get `insertL-f` or `insertR-f`.

```
(define seqL
  (lambda (new old l)
    (cons new (cons old l))))

(define seqR
  (lambda (new old l)
    (cons old (cons new l))))
```

- Time to write the left or write agnostic function which will take `seqL` or `seqR`. Notice we remove the `test?` this is so because are sort of removing a degree of freedom. We are abstracting out the part that varies to make functions more general and in this case we are 'okay' with `eq?`.

```
(define insert-g
  (lambda (seq)
    (lambda (new old l)
      (cond
        ((null? l) (quote ()))
```

```
        ((eq? (car l) old)
         (seq new old (cdr l)))
        (else (cons (car l)
                    ((insert-g seq) new old (cdr l))))))))
```

- Now we can define the left and right functions by passing helper functions appropriately. But do we need to pass `seqL` and `seqR` and why not something generic as `seq`?

```
(define insertL (insert-g seqL))
(define insertR (insert-g seqR))
```

- Now comes the idea of passing anonymous or lambdas as the programming world calls it to functions. In Prof Touretzky's book also we have come across this.

```
    (define insertL-anon
      (insert-g
       (lambda (new old l)
        (cons new (cons old l)))))
```

```
> (insertL-anon 'x 'a '(a b c d))
'(x a b c d)
```

- Similarly for the `subst` substitute function we could create another helper function.

```
(define seqS
  (lambda (new old l)
    (cons new l)))

(define subst-def2
  (insert-g seqS))

(define subst-anon
  (insert-g
   (lambda (new old l)
     (cons new l))))
```

13.1.4 The Power of Functional Abstraction

- Now we come across a really smart abstraction. The book builds the `rember` function using `insert-g` and a new helper function `seqrem`. The catch is that the 'new' is simply ignored. The authors use `#f` in place of new and this is not for the lisp evaluator to look but we humans!

```
(define rember-helper
  (lambda (a l)
    ((insert-g seqrem) #f a l)))

(define seqrem
  (lambda (new old l)
    l))
```

```
> (rember-helper 'fries '(burger fries pizza chips))
'(burger pizza chips)
```

- What we have just seen is the power of abstraction!
- Another example is the erstwhile `value` function where again we have common `cond` lines in the function definition.

```
(define value-helper
  (lambda (nexp)
    (cond
      ((atom? nexp) nexp)
      ((eq? (operator nexp) (quote +))
       (o+ (value-helper (1st-sub-exp nexp))
           (value-helper (2nd-sub-exp nexp))))
      ((eq? (operator nexp) (quote x))
       (o* (value-helper (1st-sub-exp nexp))
           (value-helper (2nd-sub-exp nexp))))
      (else
       (o^ (value-helper (1st-sub-exp nexp))
           (value-helper (2nd-sub-exp nexp)))))))
```

- Here we can abstract away the common portions of the function by writing a helper function `atom-to-function`.

```
(define atom-to-function
  (lambda (x)
    (cond
      ((eq? x (quote +)) o+)
      ((eq? x (quote x)) o*)
      (else o^))))
```

```
> (atom-to-function '+)
#<procedure:o+>
> ((atom-to-function '+) 5 3)
8
```

Note that the `1st-sub-exp` and `2nd-sub-exp` are just names and need to be given definitions for these functions of `value` and its variants to run. The code shown in the book is to only illustrate abstraction and helper functions.

- Here we should clarify some terms and reiterate them:
  - Accessor Functions are also called Helper Functions. These functions help access some part or element in a list.
  - Meta Functions are those functions which operate on functions themselves. Meta Functions take in functions and return functions.
  - We can have a combination of accessor functions in a meta function.
- This chapter is taking us the readers through this cycle essentially:
  - Writing a 'crude' or basic function which just gets the job done for instance the `insertL` function.
  - Then we go to a factored or parametrized version of the 'crude' function now called 'f' function such as `insertL-f`.
  - From the factored version we go to a general version or the 'g' version such as ~insert-g'.
  - And finally we will reach the evaluator example `value-helper` above.
- We had written the `multirember` function earlier which was fairly easy.

```
(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)))))))
```

- Now the factorized version is below.

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

```
> ((multirember-f eq?) 'samosa '(pakora samosa chaat jalebi samosa chutney))
'(pakora chaat jalebi chutney)
```

- We can now make the test in `multirember-f` be only `eq?` by the following definition:

```
(define multirember-eq?
  (multirember-f eq?))
```

```
> (multirember-eq? 'tuna '(tuna fish is good for health but not fried tuna fish))
'(fish is good for health but not fried fish)
```

- Next line of reasoning is that the `multirember-f` gos through the `lat` looking only for tuna and using `eq?` to test that. So we can probably merge this as one function. So let us write a generic function which uses `eq?` to test if it is tuna or not.

```
(define eq?-tuna
  (eq?-c (quote tuna)))
```

```
> (eq?-tuna 'fish)
#f
> (eq?-tuna 'tuna)
#t
```

- Now we can write a function `multiremberT` which takes `eq?-tuna` as an input for the test and comparison to tuna and a list of atoms as the second argument. This makes our function compact and purposeful. Taking a step back `multiremberT` does the following:
  - Takes two arguments: one is a function and another is a list. We could perhaps curry this further and have just one argument (perhaps).
  - The functional argument is a combination of another function `eq?-tuna` which in turn calls `eq?-c` ()itself is a curried function) with the symbol tuna.
  - The basic recursion is fairly simple. If we see `multiremberT` takes a function calls another function which in turn calls another function. These are layers of abstractions for a higher order function.

  Let us write the function now.

```
(define multiremberT
  (lambda (test? lat)
    (cond ((null? lat) (quote ()))
          ((test? (car lat))
           (multiremberT test? (cdr lat)))
          (else (cons (car lat)
                      (multiremberT test? (cdr lat)))))))
```

```
> (multiremberT eq?-tuna '(shrimp salad tuna salad and tuna))
'(shrimp salad salad and)
```

13.1.5 Collector Functions or Continuations

- This is the last commandment for the book and it gets fairly dense.
- We write a function called `multirember&co`. The book will now slowly step by step explain the definition of the function. But we will state it first and then work through it to understand what it really is doing.

```
(define multirember&co
  (lambda (a lat col)
    (cond ((null? lat) (col (quote ()) (quote ())))
          ((eq? (car lat) a)
           (multirember&co a (cdr lat)
                           (lambda (newlat seen)
                             (col newlat
                                  (cons (car lat) seen)))))
          (else
           (multirember&co a (cdr lat)
                           (lambda (newlat seen)
                             (col (cons (car lat) newlat)
                                  seen)))))))
```

- Another function is defined which will be used as the continuation or collectors as we progress. It is called right now `a-friend`.

```
(define a-friend
  (lambda (x y)
    (null? y)))
```

13.1.6 Continuation Passing Style

- In this section we blow the lid off. Here we will learn a foundational concept for interpreters, compilers, lazy evaluation, backtracking etc.
- Now let us get to work on `multirember&co` function with the null list `()` as `lat`. We pass `a` as `'tuna`, `lat` as `()` and the `col` that is the collector as `a-friend`. Since the `lat` is empty it hits the first line of `cond` which is `((null? lat) (col (quote ()) (quote ())))`. The collector takes the two `quote()` and since the second one is an empty list it returns a `#t`. Thus this comibination of arguments when passed to `multirember&co` will return `#t`.

```
(define multirember&co
  (lambda (a lat col)
    (cond ((null? lat) (col (quote ()) (quote ())))
          ((eq? (car lat) a)
           (multirember&co a (cdr lat)
```

```
                          (lambda (newlat seen)
                            (col newlat
                                 (cons (car lat) seen)))))
          (else
           (multirember&co a (cdr lat)
                          (lambda (newlat seen)
                            (col (cons (car lat) newlat)
                                 seen)))))))

(define a-friend
  (lambda (x y)
    (null? y)))
```

- The question with the input of `lat` as `(strawberries tuna and swordfish)` should not even be attempted right now. After this section the answer will come in an instant. The authors perhaps should have simplified the whole section a little bit more.

- When `lat` is `(tuna)` and the collector is `a-friend`, we hit the second `cond` test. The recursive call to `multirember&co` is with `('tuna '() col1)`. Here `col1` is the lambda function given as:

  ```
  (lambda (newlat seen) (col newlat (cons (car lat) seen)))
  ```

  But in `col1` the `col` refers to the original `a-friend`. The book calls this `col1` as `new-friend`.

  ```
  (define new-friend
    (lambda (newlat seen)
      (a-friend newlat (cons (quote tuna) seen))))
  ```

  When we recur on `lat` as `()` then the the first `cond` gets hit. Here the `col` is called the `new-friend`. So we pass two blank lists to `new-friend`. This in turn calls `a-friend` with a blank list and a `(tuna)`. This returns `#f`. In plain english what this whole mechanism is doing is "Remove all `'tuna` from `(and tuna)`, and then ask: did we remove none? We did remove one, so the answer is `#f`."

- Now if we think of the first question where the lat is `(strawberries tuna and swordfish)` we know the answer will be `#f` but let us wait for the next example where `lat` is `(and tuna)`.
- Let us write down all the calls for `(and tuna)`.

  First Call

  Arguments:

  `a` = `'tuna`

  `lat` = `'(and tuna)`

  `col` = `a-friend`

  `lat` is not null, and `(car lat)` is `'and`, which is not `a`, so we go to the `else` branch:

  This collector is called `latest-friend` in the book.

```
(lambda (newlat seen) (a-friend (cons 'and newlat) seen))
```

  Second Call

  Arguments:

  `a` = `'tuna`

  `lat` = `'(tuna)`

  `col` = `latest-friend`

  `lat` is not null, and `(car lat)` is `'tuna`, so we go to the middle branch:

  This collector is our already described `new-friend` earlier but the `col` is actually the `latest-friend`.

```
(lambda (newlat seen) (latest-friend newlat (cons 'tuna seen)))
```

  Third Call

  Arguments:

  `a` = `'tuna`

  `lat` = `'()`

  `col` = `new-friend` from above where `col` is `latest-friend`

  `lat` is null, we hit the first condition of the `cond`:

```
(new-friend '() '())
```

Unwinding Collectors

This is where we see the inte-relationships of the various collector functions.

```
(new-friend '() '())
(latest-friend '() (cons 'tuna '()))
(latest-friend '() '(tuna))
(a-friend '(and) '(tuna))
(null? '(tuna))
> #f
```

We can see the book calls out the last step where they say the arguments to `a-friend` is `'(and)` and `'(tuna)` which is exactly what we got in out detailed flow above. What we notice is that the atoms which are not collected are in one list and the atoms which are collected are in the other list.

- If we change the collector function to say return the length of the list where we have not collected then we will get the count of the list where the `a` is not present. The book gives this collector as an example.

```
(define last-friend
  (lambda (x y)
    (length x)))
```

13.2 Examples on Continuations

13.2.1 Inserting and counting: `multiinsertLR&co`

- We revisit the functions `multiinsertL` and `multiinsertR` from chapter 3: Cons the magnificent.
- We can combine the left and right functions as one below:

```
(define multiinsertLR
  (lambda (new oldL oldR lat)
```

```
(cond
  ((null? lat) (quote ()))
  (eq? (car lat) oldL
       (cons new
             (cons oldL
                   (multiinsertLR new oldL oldR (cdr lat)))))
  (eq? (car lat) oldR
       (cons oldR
             (cons new
                   (multiinsertLR new oldL oldR (cdr lat)))))
  (else
   (cons (car lat)
         (multiinsertLR new oldL oldR (cdr lat)))))))
```

- Now we can write a `multiinsertLR&co` which will have a collector function. The collector will work on the new list of atoms and on the number of left and right insertions. The outline of the function looks something like this.

```
(define multiinsertLR&co
  (lambda (new oldL oldR lat col)
    (cond
      ((null? lat) (col (quote ()) 0 0))
      ((eq? (car lat) oldL)
       (multiinsertLR&co new oldL oldR (cdr lat)
                         (lambda (newlat L R)
                           ...)))
      ((eq? (car lat) oldR)
       (multiinsertLR&co new oldL oldR (cdr lat)
                         (lambda (newlat L R)
                           ...)))
      (else
       (multiinsertLR&co new oldL oldR (cdr lat)
                         (lambda (newlat L R)
                           ...))))))
```

- The base collector in the `(null? lat)` test returns an empty list thus

we return `(quote ())` and since there are no occurrences of `oldL` or `oldR` the count remains 0 for both these.

- The book now asks us exactly what I mention above that what happens in the case we pass an empty list as the `lat`.

- The `else` part of the `cond` gets triggered when `(car lat)` is neither equal to `oldL` or `oldR`. The collector in the `else` part will take this `newlat` which is essentially produced by `multiinsertLR&co` on `(cdr lat)`, `oldL`, and `oldR`. The second and third arguments are the left and right insertion counts.

- This `else` collector will apply the collector function on a `cons` of `(car lat)` to `newlat` and that is `(cons (car lat) newlat)` and the other two arguments to collector will be `L` and `R` because we need to keep a count of the insertions.

- The two collector in the `cond` for `oldL` and `oldR` will essentially be an appropriate `cons` of `new` to `oldL` or `oldR`. In addition we will increment `oldL` or `oldR` appropriately. So finally we get `multiinsertLR&co` as below and also we write the collector function here.

```
(define multiinsertLR&co
  (lambda (new oldL oldR lat col)
    (cond
      ((null? lat) (col (quote ()) 0 0))
      ((eq? (car lat) oldL)
       (multiinsertLR&co new oldL oldR (cdr lat)
                         (lambda (newlat L R)
                           (col (cons new
                                      (cons oldL newlat))
                                      (add1 L) R))))
      ((eq? (car lat) oldR)
       (multiinsertLR&co new oldL oldR (cdr lat)
                         (lambda (newlat L R)
                           (col (cons oldR
                                      (cons new newlat))
                                L (add1 R)))))
      (else
       (multiinsertLR&co new oldL oldR (cdr lat)
                         (lambda (newlat L R)
```

```
                                    (col (cons (car lat) newlat) L R)))))))
```

```
    (define col
      (lambda (newlat L R)
        (cons newlat (cons L (cons R (quote ()))))))
```

```
> (multiinsertLR&co 'salty 'fish 'chips
                    '(chips and fish or fish and chips) col)
'((chips salty and salty fish or salty fish and chips salty) 2 2)
```

13.2.2 Multiple Number Operations: `evens-only*&co`

- We got back to the `*` functions in this example. `*` functions basically traverse entire nested lists.
- Short term objective is to right a functions which removes all odd numbers from a nested list of numbers. First we can write a predicate which tests if a number is even or not.

```
(define even?
  (lambda (n)
    (cond
      ((= (modulo n 2) 0) #t)
      (else #f))))

(define evens-only*
  (lambda (l)
    (cond
      ((null? l) (quote ()))
      ((atom? (car l))
       (cond
         ((even? (car l))
          (cons (car l) (evens-only* (cdr l))))
         (else (evens-only* (cdr l)))))
      (else (cons (evens-only* (car l))
                  (evens-only* (cdr l)))))))
```

- Note that I have used a different way to test whether a number is even or not from that of in the book. We should also look at the `trace` of

this `evens-only*` to refresh the stack flow of such a function.

```
> (evens-only* '((9 1 2 8) 3 10 ((9 9) 7 6) 2))
>(evens-only* '((9 1 2 8) 3 10 ((9 9) 7 6) 2))
> (evens-only* '(9 1 2 8))
> (evens-only* '(1 2 8))
> (evens-only* '(2 8))
> >(evens-only* '(8))
> > (evens-only* '())
< < '()
< <'(8)
< '(2 8)
> (evens-only* '(3 10 ((9 9) 7 6) 2))
> (evens-only* '(10 ((9 9) 7 6) 2))
> >(evens-only* '(((9 9) 7 6) 2))
> > (evens-only* '((9 9) 7 6))
> > >(evens-only* '(9 9))
> > >(evens-only* '(9))
> > >(evens-only* '())
< < <'()
> > >(evens-only* '(7 6))
> > >(evens-only* '(6))
> > > (evens-only* '())
< < < '()
< < <'(6)
< < '(() 6)
> > (evens-only* '(2))
> > >(evens-only* '())
< < <'()
< < '(2)
< <'((() 6) 2)
< '(10 (() 6) 2)
<'((2 8) 10 (() 6) 2)
'((2 8) 10 (() 6) 2)
```

- We write the function `evens-only*&co` which removes the odd numbers from a nested list, multiplies all the even numbers and sums all the odd numbers which had been removed and gives this list as the answer.

```
(define evens-only*&co
  (lambda (l col)
    (cond
      ((null? l) (col (quote ()) 1 0))
      ((atom? (car l))
       (cond
         ((even? (car l))
          (evens-only*&co (cdr l)
                          (lambda (newl p s)
                            (col (cons (car l) newl)
                                       (* (car l) p) s))))
         (else (evens-only*&co (cdr l)
                               (lambda (newl p s)
                                 (col newl p
                                           (+ (car l) s)))))))
      (else (evens-only*&co
             (car l) (lambda (al ap as)
                       (evens-only*&co
                        (cdr l) (lambda (dl dp ds)
                                  (col (cons al dl)
                                       (* ap dp)
                                       (+ as ds)))))))))))
```

```
> (evens-only*&co '((9 1 2 8) 3 10 ((9 9) 7 6) 2) col)
'(((2 8) 10 (() 6) 2) 1920 38)
```

- The last collector function we write in this chapter is not coincidentally called `the-last-friend`.

```
(define the-last-friend
  (lambda (newl product sum)
    (cons sum (cons product newl))))
```

When we pass this to `evens-only*&co` we get:

```
> (evens-only*&co '((9 1 2 8) 3 10 ((9 9) 7 6) 2) the-last-friend)
'(38 1920 (2 8) 10 (() 6) 2)
```

Core Terms/Concepts Learnt

- Higher Order Functions
- Meta Functions
- Functions taking other functions as arguments
- Functions returning functions
- Collectors or Continuations
- Helper Functions
- Continuation Passing Style

14 ... and Again, and Again, and Again, ...

The hardest chapter: Derive recursion without names using self-application. Build the conceptual machinery that leads to fixed point combinators and the Y combinator.

Key Takeaway:
Partial Functions vs Total Functions

Deriving the Y Combinator

Y Combinator

14.1 Partial Functions

- With a series of examples we are led to the concept of partial functions. Initially the reader will be stumped to find that their elementary logic be defied but on reading what the function `looking` and `keep-looking` do then things fall into place easily. In this `lat` which is the argument to `looking` the numbers determine where we eventually land up after multiple recursions and at times we will not reach the desired destination conclusively hence the function will return `#f` while at other times it could skip certain elements in `lat` forever. When the recursion skips certain element(s) then it is said to be a partial function. This is a very crude definition from what we observe from the initial examples.

- The functions can be written as following. Note the argument `sorn` stands for symbol or number.

```
(define looking
  (lambda (a lat)
    (keep-looking a (pick 1 lat) lat)))

(define keep-looking
  (lambda (a sorn lat)
    (cond ((number? sorn)
           (keep-looking a (pick sorn lat) lat))
          (else (eq? sorn a)))))
```

- The unusual thing about the function `keep-looking` is that it does not recur on a part of the list and the book calls it 'unnatural' recursion.
- So depending on how the `lat` argument to `keep-looking` is the result returned could be anything. Sometimes (as above) it would skip certain elements forever, at times the recursion will never stop (when a tuple of numbers is provided as `lat`). Example of an endless loops can be `(7 2 4 7 5 6 3)` and `(7 1 2 caviar 5 6 3)`.
- Now the book calls `keep-looking` a partial function.
- An example of infinite recursion or as the book calls it the 'most partial' function.

```
(define eternity
  (lambda (x)
    (eternity x)))
```

- Now we come to the function `shift`. This function takes a pair in which the first element is itself a pair. It then takes the second of the first pair and builds a new pair with this second as the first and the second pair as the second element of the second new pair. Function definition ans examples below.

```
    (define shift
      (lambda (pair)
        (build (first (first pair))
               (build (second (first pair))
                      (second pair)))))
> (shift '((a b) c))
'(a (b c))
> (shift '((a b) (c d)))
'(a (b (c d)))
```

- The next function `align` is tricky. It takes an argument called `pora` which is a pair or an atom. It aims to normalize a nested pair into a right associated form where the left branch is always atomic. It converts `pora` to a form like this `(atom.fully aligned tail)`.

```
(define align
  (lambda (pora)
    (cond ((atom? pora) pora)
          ((a-pair? (first pora))
           (align (shift pora)))
          (else (build (first pora)
                       (align (second pora)))))))
```

- We want to now count the number of elements in `pora`. This is done by the function `length*` (`*` because it will traverse the entire nested list).

```
(define length*
  (lambda (pora)
    (cond ((atom? pora) 1)
          (else
           (+ (length* (first pora))
              (length* (second pora)))))))
```

- The `length*` function gives equal weight to all atoms to the arguments of `pora` and does not consider the complexity of the structure of `pora`. To correct this we write a new function called `weight*`.

```
(define weight*
  (lambda (pora)
    (cond ((atom? pora) 1)
          (else
           (+ (* (weight* (first pora)) 2)
              (weight* (second pora)))))))
```

- The example given is important and we can work it out by hand. We notice the weights of the arguments get simpler once aligned.

  Before aligning:

```
((a b) c)
```

```
= first: (a b)
  second: c

  (weight* (a b)) = (+ (*1 2) 1) = 3
  (weight* c)     = 1

Total = 3*2 + 1 = 7
```

After aligning:

```
(a (b c))

= first: a
  second: (b c)

  (weight* a)     = 1
  (weight* (b c)) = (+ (*1 2) 1) = 3

Total = 1*2 + 3 = 5
```

- `align` is not a partial function because it yields a value for every argument.
- We write `shuffle` which is also a partial function. What this function does it make the first element of the `pora` an atom and pushes the complex parts to the second element. Why it is partial is because if the argument is `((a b) (c d))` this function will go on till eternity.

```
(define shuffle
  (lambda (pora)
    (cond
      ((atom? pora) pora)
      ((a-pair? (first pora))
       (shuffle (revpair pora)))
      (else (build (first pora)
                   (shuffle (second pora)))))))

> (shuffle '(a (b c)))
```

```
'(a (b c))
> (shuffle '((a b) (c)))
'((c) (a b))
> (shuffle '((a b) c))
'(c (a b))
> (shuffle '((a b) (c d)))
. . user break
```

14.2 Recursive nature of functions

- Next is the famous Collatz Conjecture which is defined as the function `C` in the book. Collatz rule is simple where we start from any positive integer and do repeated procedure basis the rule: if even divide by 2 and if odd then multiply by 3 and add 1. No matter what the sequence terminates in 1. This has not been proven yet rigorously in mathematics since 1937. Paul Erdős said about the Collatz conjecture that *Mathematics may not be ready for such problems.* Why the book shows this example is because the computation of `C` is not narrowing down to a number. The values of interim `C` jump around. I have added a few examples.

```
(define C
  (lambda (n)
    (cond
      ((one? n) 1)
      (else
       (cond
         ((even? n) (C (/ n 2)))
         (else (C (add1 (* 3 n)))))))))
```

```
> (C 2)
>(C 2)
>(C 1)
<1
1

> (C 3)
>(C 3)
>(C 10)
```

```
>(C 5)
>(C 16)
>(C 8)
>(C 4)
>(C 2)
>(C 1)
<1
1

> (C 11)
>(C 11)
>(C 34)
>(C 17)
>(C 52)
>(C 26)
>(C 13)
>(C 40)
>(C 20)
>(C 10)
>(C 5)
>(C 16)
>(C 8)
>(C 4)
>(C 2)
>(C 1)
<1
1
>
```

- The next is the Ackermann function. The book uses the common version of this called the Ackermann–Péter function. Ackermann function is a total function but is not primitive recursive. This means a computer program cannot compute it always generally speaking. Let us write the function and give it few seed arguments.

```
(define A
  (lambda (n m)
    (cond
      ((zero? n) (add1 m))
```

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

> (A 2 1)
>(A 2 1)
> (A 2 0)
> (A 1 1)
> >(A 1 0)
> >(A 0 1)
< <2
> (A 0 2)
< 3
>(A 1 3)
> (A 1 2)
> >(A 1 1)
> > (A 1 0)
> > (A 0 1)
< < 2
> >(A 0 2)
< <3
> (A 0 3)
< 4
>(A 0 4)
<5
5
```

Now if we pass the argument (4 3) my machine will go on endlessly probably till the end of time.

14.3 Turing Halting Problem

- The next endeavor is to write a function which will tell us if a function is a total or a partial function. The book is slowly leading us into the true definition of recursion and functions in computer science. This is the Turing Halting Problem. Some functions can be described but not defined. No function in Scheme (or any programming language) can determine in general whether another function halts. This is not possible in Mathematics. There is no general procedure that can always tell whether a computation will halt. This is the cornerstone of computability theory, limits of programming languages limits of mathematics, Gödel incompleteness, and Turing machines.

14.4 What is Recursion? What is a Function?

- `will-stop?` is a function to show the Turing Halting problem cannot be 'defined' in our programming language or for that matter any programming language. This is the last leg of this chapter.

- We revisit the function `length` which counts the number of S-expressions in a list. It was in the chapter 'Number Games'.

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

- The last line of `length` refers to itself so if we do not `define length` then how would we refer back to it in the recursive call?

14.4.1 Closures and Lexical Scoping

- The first time we have a `lambda` expression without defining it. Dr. Racket will see the `lambda` expression, evaluate it and then return a procedure value which is a closure. So the value of a `lambda` expression is a procedure. Now what is a closure? A closure is a function in its lexical scoped environment. A closure is a pair of `(code environment)`. We can use an example here to differentiate between lexical scoping and dynamic scoping.

```
(let ((x 5))
  (lambda (y) (+ x y)))
```

In this the `lambda` expression remembers the value `x = 5` even after it has left the `let`. In lexical scoping a variable's meaning is determined by where it appears in the source code. In dynamic scoping a variable's meaning is determined by which function called which function at runtime.

```
(define x 10)

(define (f)
  x)

(define (g)
  (let ((x 20))
    (f)))
```

Under lexical scoping what does `(g)` see the value of `x`? `g` calls `f` and `f` in turn sees that `x = 10`. Thus it is checking the environment to get the value of `x`. But under dynamic scoping `(g)` will check the value of `x` on the call stack. And the value of `x` on call stack has changed from 10 to 20. So in this case `x = 20`.

14.4.2 The Y Combinator Derivation

- Back to the first `lambda` expression. Here it is.

```
(lambda (l)
  (cond
    ((null? l) 0)
    (else (add1 (eternity (cdr l))))))

#<procedure:...chemer-workbook.rkt:907:0>
```

- If we call this for an empty list all is good we get `0` as return. But when we call it for any other list the `eternity` call will go on for eternity! It keeps calling it itself with `'()`.

```
#<procedure:...chemer-workbook.rkt:906:0>
#<procedure:...chemer-workbook.rkt:912:3>
> ((lambda (l)
  (cond
    ((null? l) 0)
    (else (add1 (eternity (cdr l))))) '())
0
> ((lambda (l)
  (cond
    ((null? l) 0)
    (else (add1 (eternity (cdr l))))) '(a))
. . user break
```

- We can call the `eternity` as `length-0` because it returns `0` for an empty list. Thus we can rewrite it as below.

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

- Now for `length-1` up to one we can nest the two lambdas. Since we cannot use `define` to put in the function we place the `lambda` object itself. In fact we can keep adding for measuring length of longer lists.

```
(lambda (l)
  (cond
    ((null? l) 0)
    (else
     (add1
      ((lambda (l)
         (cond
           ((null? l) 0)
           (else (add1
                  (eternity (cdr l))))))
      (cdr l))))))
```

The same nesting can be done for `length-2` showing an application on a 2 element list below.

```
> ((lambda (l)
  (cond
    ((null? l) 0)
    (else
     (add1
      ((lambda (l)
         (cond
           ((null? l) 0)
           (else (add1
                  ((lambda (l)
                     (cond
                       ((null? l) 0)
                       (else
                        (add1
                         (eternity
                          (cdr l))))))
                   (cdr l))))))
       (cdr l)))))) '(a b))
2
```

- For `length-2` we can add more nesting (as above). I would not pass the function for multiple examples. One thing to note is that the scope of the parameters is local and we see the term `length` being used across. I think this is done deliberately by the authors to drive this point home.
- The playful but insightful question to ask is 'What is recursion?' now.
- If we could go on and on and keep nesting `lambdas` up to infinity with `length-infinity` being the last one perhaps then we would have derived the function for `length`. But it is not possible to write an infinite function. But we do see patterns emerging, repetitions. We can abstract out the common recurring patterns as another function which is the ninth commandment.

```
((lambda (f)
   (lambda (l)
     (cond
```

```
      ((null? l) 0)
      (else (add1 (f (cdr l)))))))
((lambda (g)
   (lambda (l)
     (cond
       ((null? l) 0)
       (else (add1 (g (cdr l)))))))
 eternity))
```

```
#<procedure:...chemer-workbook.rkt:906:0>
#<procedure:...chemer-workbook.rkt:912:3>
#<procedure:...chemer-workbook.rkt:919:3>
> (((lambda (f)
    (lambda (l)
      (cond
        ((null? l) 0)
        (else (add1 (f (cdr l)))))))
  ((lambda (g)
     (lambda (l)
       (cond
         ((null? l) 0)
         (else (add1 (g (cdr l)))))))
   eternity)) '())
0
> (((lambda (f)
    (lambda (l)
      (cond
        ((null? l) 0)
        (else (add1 (f (cdr l)))))))
  ((lambda (g)
     (lambda (l)
       (cond
         ((null? l) 0)
         (else (add1 (g (cdr l)))))))
   eternity)) '(a))
1
> (((lambda (f)
    (lambda (l)
      (cond
```

```
                    ((null? l) 0)
                    (else (add1 (f (cdr l)))))))
                ((lambda (g)
                   (lambda (l)
                     (cond
                       ((null? l) 0)
                       (else (add1 (g (cdr l)))))))
                 eternity)) '(a b))
               . . user break
```

- For `length-2` the change is below.

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

> (((lambda (length)
    (lambda (l)
      (cond
        ((null? l) 0)
        (else (add1 (length (cdr l)))))))
  ((lambda (length)
     (lambda (l)
       (cond
         ((null? l) 0)
         (else (add1 (length (cdr l)))))))
   eternity))
. . user break
```

```
  ((lambda (length)
     (lambda (l)
       (cond
         ((null? l) 0)
         (else (add1 (length (cdr l)))))))
   eternity))) '(a b))
2
```

- We still have repetitions in our functions above. We can still abstract away and make functions.
- The authors guide us to rename the function which takes `length` as an argument and returns a function that looks like length.
- Now comes a syntactic trick. Assume that the function to generate or get length is called `F`:

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

- We have to pass `eternity` to it. So we have to something like `F eternity`. This is the structure. An outer `lambda` can take this `F` and apply that `F` to eternity. Let us build a basic length of empty list as an example.

```
((lambda (mk-length)
   (mk-length eternity))
 (lambda (length)
   (lambda (l)
     (cond
       ((null? l) 0)
       (else (add1 (length (cdr l))))))))
```

What we have done is have `((lambda (F) (F eternity)).....F)`. The outer `lambda` returns `F eternity` just like before! Why are we doing this? So that we can nest it.

- Now if we want to go for `length-1` or `length-2` we just need to make the outer `lambda` pass more `mk-length`. For instance:

```
((lambda (mk-length)
   (mk-length
    (mk-length
     (mk-length
      (mk-length
       (mk-length eternity))))))
 (lambda (length)
   (lambda (l)
     (cond
       ((null? l) 0)
       (else (add1 (length (cdr l))))))))
```

- The next manipulation which the book does is to get a structure of this form.

```
((lambda (mk-length)
   (mk-length mk-length))
 (lambda (mk-length)
   (lambda (l)
     (cond
       ((null? l) 0)
       (else (add1
              ((mk-length eternity) (cdr l))))))))
```

  The outer `lambda` is there and an inner `lambda`. After applying the inner `lambda` to the outer one we essentially get inner being applied to inner. For testing `'()` we see that it hits `null?` and actually does not go into recurrence.

- Also note that parameter name in the function does not matter.

- The example for `(apples)` yields 1. This is something on the lines of `(F (F '(apples)))` where `F` is a combination of the outer `lambda` and inner one.

- We can keep passing `mk-length` to itself.

```
((lambda (mk-length)
   (mk-length mk-length))
 (lambda (mk-length)
   (lambda (l)
     (cond
       ((null? l) 0)
       (else (add1
              ((mk-length mk-length) (cdr l))))))))
```

- We extract `mk-length` call to itself but we get an infinite recursion actually.

```
    ((lambda (mk-length)
       (mk-length mk-length))
     (lambda (mk-length)
       ((lambda (length)
          (lambda (l)
            (cond
              ((null? l) 0)
              (else (add1 (length (cdr l)))))))
        (mk-length mk-length))))
```

```
#<procedure:...chemer-workbook.rkt:1027:3>
. Interactions disabled; out of memory
```

The book shows that with next set of recursions as examples.

- Actually `mk-length` now is not returning a function anymore (it is a list).
- The book now manipulates it into another lambda function.

```
((lambda (mk-length)
   (mk-length mk-length))
 (lambda (mk-length)
   (lambda (l)
     (cond
       ((null? l) 0)
       (else (add1
```

```
              ((lambda (x)
                 ((mk-length mk-length) x))
               (cdr l))))))))
```

```
((lambda (mk-length)
   (mk-length mk-length))
 (lambda (mk-length)
   ((lambda (length)
      (lambda (l)
        (cond
          ((null? l) 0)
          (else (add1 (length (cdr l)))))))
    (lambda (x)
      ((mk-length mk-length) x)))))
```

- The next modification is to move the `(lambda (length))` out. Before we did the opposite by replacing a name with its value, now we extract the value and give it a name. We actually get the Y combinator now but the book will separate out the `length` function later.

```
((lambda (le)
   ((lambda (mk-length)
      (mk-length mk-length))
    (lambda (mk-length)
      (le (lambda (x)
            ((mk-length mk-length) x))))))
 (lambda (length)
   (lambda (l)
     (cond
       ((null? l) 0)
       (else (add1 (length (cdr l))))))))
```

- Extracting out the `length` function and giving it the official nomenclature.

```
(define Y
```

```
(lambda (le)
  ((lambda (f) (f f))
   (lambda (f)
     (le (lambda (x) ((f f) x))))))
```

- `(Y Y)` will be infinite recursion. `Y` is called the Y Combinator, `le` is the little evaluator. This `(Y Y)` has no base case.
- This is a very difficult chapter and the authors could have done a much better job in teaching this. Trying to be clever does not lead to clarity in this chapter at all. The above were my raw notes, I am going to put diagrams and more simplification in terms of visuals to clarify this. The intent of the chapter should be clear after reading these notes but if one were to derive the Y Combinator by themselves it will be a tough task.

Core Terms/Concepts Learnt

- Partial Functions
- Total Functions
- Combinators
- Turing Halting Problem
- Fixed Point Combinators
- Collectors or Continuations
- Y Combinator

15 What Is the Value of All of This?

Climax of the book which leads to getting ready to build an interpreter.

Key Takeaway:
Use all previous ideas to sketch an interpreter: environments, lookup, closures, and application. Shows that evaluation itself is just another recursive function built from structural principles.

15.1 Entries and Lookups

- An entry is defined as a pair of lists whose first list is a set and both the lists in the entry have equal length.
- We seem to be defining a method to look for a particular element in the set get its position and then lookup in the second list the element in the same position.
- Now we build a function to do exactly this by looking up the position of an input in the first set and then finding the element in the second list.

```
(define lookup-in-entry
  (lambda (name entry entry-f)
    (lookup-in-entry-help name
                          (first entry)
                          (second entry)
                          entry-f)))

(define lookup-in-entry-help
  (lambda (name names values entry-f)
    (cond
      ((null? names) (entry-f name))
      ((eq? (car names) name)
       (car values))
      (else (lookup-in-entry-help name
                                  (cdr names)
                                  (cdr values)
                                  entry-f)))))
```

- We have a helper function `lookup-in-entry-help`. The book always uses fallback functions so the code stays 'pure lambda' - no errors, no special forms, no exceptions.
- This is how the interpreter will find values defined in nested scopes. We are building an interpreter in this chapter perhaps.
- We will stack a new entry on top of the existing table via `(define extend-table ...)`.
- So now we have a list of entries inside a table. We can look for specific values and if the value is appearing multiple times the evaluation order makes sure to return the first instance only.
- The `lookup-in-table` function is written which is fairly simple. The note here is how the `cdr` portion of the table is being recursed into. The bottom `lambda` function does that.

```
(define lookup-in-table
  (lambda (name table table-f)
    (cond
      ((null? table) (table-f name))
      (else (lookup-in-entry name (car table)
                             (lambda (name)
                               (lookup-in-table name
                                                (cdr table)
                                                table-f)))))))
```

- Reiterating that `value` is the function that returns the natural value of expressions.

```
> (car (quote (a b c)))
'a
```

- Then we `cons` in a nested fashion. Fairly straightforward.

```
(cons rep-a
      (cons rep-b
            (cons rep-c (quote ()))))
```

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

- The book is driving us to something similar we see in macros of Common Lisp. We are quoting `car` and `quote` and then that is getting used to build a Lisp object which can further be evaluated.

```
    (cons rep-car
          (cons (cons rep-quote
                      (cons
                       (cons rep-a
                             (cons rep-b
                                   (cons rep-c
                                         (quote ()))))
                       (quote ())))
          (quote ())))

> (cons 'car
      (cons (cons 'quote
                  (cons
                   (cons 'a
                         (cons 'b
                               (cons 'c
                                     (quote ()))))
                   (quote ())))
      (quote ())))
'(car '(a b c))
```

15.2 Values

- Next is a series of expressions we can test out in the REPL of Dr Racket.

```
> (car (quote (a b c)))
'a
```

```
> (car (quote (a b c)))
'a

> (quote (car (quote (a b c))))
'(car '(a b c))

> (value (add1 6))
7

> (value 6)
6

> (value (quote nothing))
'nothing
```

- Although the `value` function which we defined this chapter assumes that it can take arguments of other types. So readers should be careful here, they will get an error. The book should have been more explicit.

```
(value '((lambda (nothing)
   (cons nothing (quote ())))
 (quote
  (from nothing comes something))))

> ((from nothing comes something))
```

- Similarly the next expression also is simple. It gives `something`.

```
((lambda (nothing)
   (cond
     (nothing (quote something))
     (else (quote nothing))))
   #t)
```

15.3 Types, Value, and Meaning

- The next set of questions just wants us to know what types are and how they are being defined in the current context. Listing them below. For

folks from C language do not get confused here! Also this type system is specific to the book and has little or nothing to do with Scheme or Dr Racket. Again the book in this chapter is cryptic and for new comers tough to decipher.

  - Number (for example 42, 007) is a `*const`
  - Boolean `#t` or `#f` is a `*const`
  - Atoms are `*const`
  - `(value car)` gives the primitive but 'car' will be treated as an atom and give a `*const`
  - `(quote nothing)` gives the type in this case as `*quote`
  - An atom that is not a primitive is treated as a variable thus is given the type `*identifier`
  - A function made using `lambda` is simply given a `*lambda` type
  - When a `lambda` expression is called it is given the type `*application`
  - The conditional `cond` is given `*cond` type

  So if we notice we have 6 types in this chapter built for us.

- Now the authors say that for a particular type an appropriate action has to be taken for us to get its value.
- Now we aim to write the function `value` and this is actually the interpreter. A very rudimentary one.
- I will work backwards unlike the book. `value` takes and argument and finds its `meaning` in a table. When it is found it looks for the action to take. A conditional decides whether we need to check if its an atom and then do a `atom-to-action` and if it is anything else then we check in the function `list-to-action` for the application type. The functions are written below.

```
(define expression-to-action
  (lambda (e)
    (cond
      ((atom? e) (atom-to-action e))
      (else (list-to-action e)))))

(define atom-to-action
```

```
  (lambda (e)
    (cond
      ((number? e) *const)
      ((eq? #t) *const)
      ((eq? #f) *const)
      ((eq? e (quote cons)) *const)
      ((eq? e (quote car)) *const)
      ((eq? e (quote cdr)) *const)
      ((eq? e (quote null?)) *const)
      ((eq? e (quote eq?)) *const)
      ((eq? e (quote atom?)) *const)
      ((eq? e (quote zero?)) *const)
      ((eq? e (quote add1)) *const)
      ((eq? e (quote sub1)) *const)
      ((eq? e (quote number?)) *const)
      (else *identifier))))

(define list-to-action
  (lambda (e)
    (cond
      ((atom? (car e))
       (cond
         ((eq? (car e) (quote quote)) *quote)
         ((eq? (car e) (quote lambda)) *lambda)
         ((eq? (car e) (quote cond)) *cond)
         (else *application)))
      (else *application))))

(define value
  (lambda (e)
    (meaning e (quote ()))))

(define meaning
  (lambda (e table)
    ((expression-to-action e) e table)))
```

- A point to call out is that by the end of this chapter we truly would not have a full fledged interpreter but rather a working knowledge of the ingredients required to make one.

- Some of the types get defined for the interpreter.

```
(define *const
  (lambda (e table)
    (cond
      ((number? e) e)
      ((eq? e #t) #t)
      ((eq? e #f) #f)
      (else (build (quote primitive) e)))))

(define *quote
  (lambda (e table)
    (text-of e)))

(define *identifier
  (lambda (e table)
    (lookup-in-table e table initial-value)))

(define initial-table
  (lambda (name)
    (car (qote ()))))

(define *lambda
  (lambda (e table)
    (build (quote non-primitive)
           (cons table (cdr e)))))
```

- Finally we see the explanation of what `cond` is. It is a series of tests with fall through as I noted it before. In Common Lisp we put a `t` and in Scheme we use `else`.

```
(define evcon
  (lambda (lines table)
    (cond
      ((else? (question-of (car lines)))
       (meaning (answer-of (car lines)) table))
      ((meaning (question-of (car lins)) table)
       (meaning (answer-of (car lines)) table))
```

```
      (else (evcon (cdr lines) table)))))

(define else?
  (lambda (x)
    (cond ((atom? x) (eq? x  (quote else)))
          (else #f))))
```

Helper functions are also defined `question-of first` and `answer-of second`. In this definition every line of the `cond` should be true.

- The book asks us to start having helper functions in place `table-of first`, `formals-of second`, and `body-of third`.
- Finally

```
(define *cond
  (lambda (e table)
    (evcon (cond-lines-of e) table)))
```

With a helper of `cond-lines-of cdr`.

15.4 Evaluation and Application

- The working example is good to follow through. Here `coffee` evaluates to `#t` and then `klatsch` gives 5 as the final answer.
- An application is a list whose `car` is a function.
- An application has to determine the meaning of all its arguments but that is not the case with special forms such as `cond` or logic ones such as `and` and `or`. This is very important.
- Now we evaluate a list via a function `evlis` which evaluates every argument. And then that is applied (we use helper functions again `function-of car` and `arguments-of cdr`).

```
(define evlis
  (lambda (args table)
    (cond
```

```
      ((null? args) (quote ()))
      (else (cons (meaning (car args) table)
                  (evlis (cdr args) table))))))
(define *application
  (lambda (e table)
    (apply
     (meaning (function-of e) table)
     (evlis (arguments-of e) table))))
```

- There are 2 types of functions - primitives and non primitives.
  - A primitive is shown as `(primitive primitive-name)`
  - A non-primitive is shown as `(non-primitive (table formals body))`. `(table formals body)` is called a closure record.

```
(define primitive?
  (lambda (l)
    (eq? (first l) (quote primitive))))

(define non-primitive?
  (lambda (l)
    (eq? (first l) (quote non-primitive))))
```

- Now we write the `apply` function

```
(define apply
  (lambda (fun vals)
    (cond
      ((primitive? fun)
       ((apply-primitive (second fun) vals))
       ((non-primitive? fun)
        (apply-closure (second fun) vals))))))
```

- The helper function `apply-primitive` and `apply-closure` are written as follows.

```
(define apply-primitive
  (lambda (name vals)
    (cond
      ((eq? name (quote cons)) (cons (first vals) (second vals)))
      ((eq? name (quote car)) (car (first vals)))
      ((eq? name (quote cdr)) (cdr (first vals)))
      ((eq? name (quote null?)) (null? (first vals)))
      ((eq? name (quote eq?)) (eq? (first vals) (second vals)))
      ((eq? name (quote atom?)) (:atom? (first vals)))
      ((eq? name (quote zero?)) (zero? (first vals)))
      ((eq? name (quote add1)) (add1 (first vals)))
      ((eq? name (quote sub1)) (sub1 (first vals)))
      ((eq? name (quote number?)) (number? (first vals))))))

(define apply-closure
  (lambda (closure vals)
    (meaning (body-of closure)
             (extend-table
              (new-entry
               (formals-of closure)
               vals)
              (table-of clsoure)))))
```

- The book says that applying a non-primitive function is the same as finding the meaning of the closure's body with its table extended by an entry of the `(formals values)`.
- In the final pages of the book the authors make us do some examples of the application of `meaning` on identifiers and constants.
- We see the statement about applying a primitive function.
- Then we see a statement about applying `cons` and which `cond` line we will hit. These last few examples seem like a starter for something major to come but would not be covered in this book.
- Finally we come to `define` we skip it. We know that we truly do not needed because we have our Y Combinator.
- The interpreter can run on the interpreter if we do the transformation with the Y combinator the book says.

- Pretty sure The Seasoned Schemer will have much greater details in the book related to interpreters.
- It is time for a banquet.

Core Terms/Concepts Learnt

- Entry
- Types definition
- Evaluation, Apply
- Interpreter

'(Bon Appétit)

16 Intermission

- Books to read in my opinion from the list based on my personal knowledge
  - The Seasoned Schemer
  - Naive Set Theory
  - How to Solve It
  - Introduction to Logic
  - Structure and Interpretation of Computer Programs
  - Alice's Adventure in Wonderland

"Now get ready for the next show"