Chapter 2: Building Abstractions with Data

Caesar X Insanium

December 19, 2023

Contents

Relevant lectures

- Lecture 2B
- Lecture 3A

1 2.1 Introduction of Data Abstraction

Chapter one focused on using elementary data and procedures to construct higher order functions and can be used to express some higher level algorithms and processes using simple functions.

However, everything that has been learned use only integers, float and ratios. Even more complex behavior require the use to computational objects that have different parts. `Compound Data` is the building of abstraction by combining data objects. Compound procedures allow for elevated procedures and is the same with compound data.

A simple example is in designing a system that allows for addition of rational numbers by acting on compound data that have a component of denominator and numerator. Designing procedures that kept track of individual primitive data would be a hassle to maintain, so components must be glued together in order to be managed.

Compound data also allow for separation of the actions and procedures that act on the data from the actual implementation and background of the compound data. `Data Abstraction` is the idea that the true nature of how data ideas are represented in the hardware is hidden from user in order for easier design and management.

Linear Combinations can be expressed as such.

```
(define (linear-combination a b x y)
  (+ (* a x) (* b y)))
```

This implementation take is 4 numbers, However, we can define using data abstraction a procedure that takes 4 anything and performs the appropriate procedures defined for the data object on the data provided. The add and mul procedures determine the data type in question and use correct procedure for addition and multiplication respectably

```
(define (linear-combination a b x y)
  (add (mul a x) (mul b y)))
```

Abstraction is a technique that can be used to manage complexity and this chapter will focus on the use of data abstraction in order to separate different sections for program.

Programming languages provide the glue for allowing this forms of abstractions. From the way that data is stored and represented in computer to expression data as nothing more than procedures on primitive data. `Closure` is the idea that a language allows for the combining of both primitive data and compound data. `Symbolic` expression is augmentation of language expressive power by arbitrary symbols as opposed to numbers in which they are not defined and called until they are used store some data.

`Generic Operations` allow for defining generic operations that can be applied to different data types and a `data oriented programming` approach. This in technique, data is the most important concept and data representations are defined separately and combined `additively`

2 2.2 Hierarchical Data and Closure Property

Relevant Lecture section start at 3A 8:00, current time 28:38

We have learned that the `cons` function can be used to build simple data representations and abstractions of which individual parts can be accessed with `car` and `cdr`. Numbers and other pairs can be combined in this method.

The closure property of cons refers to its ability to represent abstract data and concepts. In accordance to specification. Hierarchical structures are made up of smaller parts coming together in order to make bigger parts.

2.1 2.2.1 Representing Sequences

The logical extension is that `cons` can be used to build arbitrary long sequences of which, lists can be built. A collection of different items that end in a nil item.

```
(define l (cons 1 (cons 2 (cons 3 (cons 3 nil)))))
```

Scheme defines this as a `list` can can be defined with a function and a series of arguments. Merely syntactic sugar for the above code segment.

```
(define 1-to-4 (list 1 2 3 4))
```

Individual elements can be accessed individually by using `car` and `cdr`.

```
(car (cdr (cdr 1-to-4)))
;; => 3
```

Certain list operations have been defined by Scheme such as `list-ref` which inputs a list and an index.

```
(define (list-ref items n)
  (if (= n 0)
      (car items)
      (list-ref (cdr items) (- n 1))))
```

Finding the length of a list is easy as well since we only need to recurse down a list until we find a null element, all the while adding one to a counter every time we recurse.

```
(define (length items)
  (if (null? items)
      0
      (+ 1 (length (cdr items)))))

(define (length-iter items)
  (define (iter a count)
    (if (null? a)
        count
        (iter (cdr a) (+ 1 count))))
  (iter items 0))
```

It is also possible to define another procedure that takes in a list and generates a new list with a new element added.

```
(define (append list1 list2)
  (if (null? list1)
      list2
      (cons (car list1) (append (cdr list1) list2))))
```

2.1.1 Mapping Over Lists

One useful function is to take a list and apply a transformation on each item and generate a new list. The scheme `map` function is for this purpose. This is a higher order procedure.

```
(define nil '())
(define (map proc items)
  (if (null? items)
      nil
      (cons (proc (car items))
            (map proc (cdr items)))))

;; Scheme Standard defines a map function that takes in a procedure of n parameters
;; and with n lists of same length
(map (lambda (x y z)
       (+ x y z))
     (list 1 2 3)
     (list 4 5 6)
     (list 7 8 9))
```

Other functions can then be defined in terms of this map function. This function would take a list of numbers and return a new list with all the numbers multiplied against another value.

```
(define (scale-list items factor)
  (map (lambda (x) (* factor x)) items))
```

The key concept here are the layers of abstraction that hides away the complexities in order to allow programmer to work on their program instead of their implementation.

Abstraction allows for using a high level concept without regards to implementation and allows and a change in implementation should not result in change in behavior for the use to deal with.

2.2 2.2.2 Hierarchical Structures

The `cons` function allows for the holding of more than just numbers, other cons boxes can hold more cons boxes. This allows for a rudimentary tree to be defined and used. Cons boxes can hold indefinite levels of cons boxes. Restricted only by the amount of memory a computer holds.

```
(cons (cons 1 2)
      (cons 3 4))
```

Tree structures lend themselves easily to recursion since operations on entire trees can be simplified to operations on branches and then to leaves. Deciding weather or not an object is a pair can be made easy with the scheme function `pair?`

```
(pair? (cons 1 2)) ; => #t
```

A simple procedure for recursively counting the number of leaves on a tree is shown.

```
(define (count-leaves x)
  (cond ((null? x) o)
        ((not (pair? x)) 1)
        (else (+ (count-leaves (car x))
                 (count-leaves (cdr x))))))
```

1. Mapping Over Trees

   The `map` procedure is a powerful concept that can be used in order to define a way to create a new list using the elements of an existing list and applying a procedure to build it. A procedure to apply the same idea to trees should not be difficult to imagine.

   ```
   ;; Here is test procedure to apply an operation accrross every object in a tree
   (define (scale-tree tree factor)
     (cond ((null? tree) nil)
           ((not (pair? tree)) (* tree factor))
           (else (cons (scale-tree (car tree) factor)
                       (scale-tree (cdr tree) factor)))))
   ```

   Then we can build a procedure that abstract away from of the details and leaves a simple interface.

   ```
   (define (tree-map proc tree)
     (cond ((null? tree) nil)
           ((not (pair? tree)) (proc tree))
           (else (cons (tree-map proc (car tree))
                       (tree-map proc (cdr tree))))))
   ```

2.3 2.2.3 Sequences as Conditional Interfaces

Conventional Interfaces are used in order to design data in a way to solve a particular problem without regards to underlying implementations. This allows for internal representation to change and as long as behavior does not change this allows for user to continue using the data with no worry.

For example given the two programs.

```
(define (even-fibs n)
  (define (next k)
    (if (> k n)
        nil
        (let ((f (fib k)))
          (if (even? f)
              (cons f (next (+ k 1)))
              (next (+ k 1))))))
  (next 0))
```

```
(define (sum-odd-squares tree)
  (cond ((null? tree) 0)
        ((not (pair? tree)) (if (odd? tree) (square tree) 0))
        (else (+ (sum-odd-squares (car tree))
                 (sum-odd-squares (cdr tree))))))
```

These follow a similar pattern in that follow similar steps

- travel through the different leaves
- selects them based on criteria
- accumulates the results

In there are steps of enumeration, mapping and accumulation. However, the different is the order in which steps are done.

1. Sequence Operations One way to think about this is laid out big the book in which each number or leave that is traversed is a signal, and they must be processed, filtered and measured in order to be useful.

   Defining signals as simply lists allow us to simply `map` over them in order to process them.

   ```
   (map square (list 1 2 3 4 5))
   ```

   Filtering can be easily implemented for lists.

   ```
   (define nums (list 1 2 3 4 5 6))

   (define (filter predicate sequence)
     (cond ((null? sequence) nil)
           ((predicate (car sequence))
            (cons (car sequence)
                  (filter predicate (cdr sequence))))
           (else (filter predicate (cdr sequence)))))

   ;; usage like so
   (filter odd? nums) ;; => (1 3 5)
   ```

   Accumulation

```
(define (accumulate op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence)
          (accumulate op initial (cdr sequence)))))

(accumulate + 0 nums);; => 21
```

Final thing need for implementation of signal processing is the enumeration for numbers and trees.

```
(define (enumurate-interval low high)
  (if (> low high)
    nil
    (cons low (enumurate-interval (+ low 1) high))))

(define (enumurate-tree tree)
  (cond ((null? tree) nil)
        ((not (pair? tree)) (list tree))
        (else (append (enumurate-tree (car tree))
                      (enumurate-tree (cdr tree))))))
```

The same procedures can now be implemented in terms of these functions. One may notice that each procedure is a sequence of operations. Designing programs in a modular and sequential way allows for easy modularity in by allowing a library of components that can then be stringed together in order to solve a problem.

```
;; Gives of the squares of fibanacchi numbers
(define (list-of-fib-square n)
  (accumulate cons nil (map square
                            (map fib (enumurate-interval 0 n)))))

;; Squares the odd elements and multiplies them together
(define (product-of-squares-of-odd-elements sequence)
  (accumulate * 1 (map square
                       (filter odd? sequence))))

;; Example on how joining these operations can be used in order to solve real
;; world problems. This reminds me of SQL selector operations
(define (salary-of-higher-paid-programmer records)
  (accumulate max 0 (map salary
                         (filter programmer? record))))
```

Moral of the story here, if one sees a low of repeating code the goal is to abstract what is possible into a modular procedure that can be called with arguments being the differentiation part of the thing.

2. Nested Mappings It is possible to use the mapping and accumulated procedures in order to device a way of implementing nested for loops. For each value of i and then for each value of j. The method for applying this is to generate a list of the relevant indexes, then mapping over and filtering relevant values and finally generate a sequence of the answers that we are looking for.

   In the example problem, we are trying to find all the unique pairs of i and j such that their sum is a prime number.

```
;; Generate pairs of indices
(define (gen-pairs n)
  (accumulate append
              nil
              (map (lambda (i)
                     (map (lambda (j)
                            (list i j))
                          (enumurate-interval 1 (- i 1))))
                   (enumurate-interval 1 n))))

(define (flatmap proc seq)
  (accumulate append nil (map proc seq)))

;; Filter Function
(define (prime-sum? pair)
  (prime? (+ (car pair) (cadr pair))))

;; Generate list with pairs and their sum
(define (make-pair-sum pair)
  (list (car pair) (cadr pair) (+ (car pair) (cadr pair))))

;; Final Generate the actual list, final answer
```

```
(define (prime-sum-pairs n)
  (map make-pair-sum (filter prime-sum?
                             (flatmap (lambda (i)
                                        (map (lambda (j) (list i j))
                                             (enumurate-interval 1 (- i 1))))
                                      (enumurate-interval 1 n)))))
```

Using nested mapping allow for easy generation of permutations and combinations. Generating permutations can be achieved with this simple procedure.

```
(define (remove item sequence)
  (filter (lambda (x) (not (= x item))) sequence))

(define (permutations s)
  (if (null? s)
      (list nil)
      (flatmap (lambda (x)
                 (map (lambda (p)
                        (cons x p))
                      (permutations (remove x s))))
               s)))

(permutations (list 1 2 3))
```

This allows use to more easily work with nested mappings so that the code the deals with the nested mapping is separate from the code the deals with generating the nested data structures that the nested maps work with.

3. 2.2.4 Example: A Picture Language We are introduced to a hypothetical picture language that makes use of the concept of a painter. If a painter is given a rectangle, it will attempt to draw an image on it given a set definitions of a rectangle and treats it as a canvas. Painters can be stacked on top of each other in a form of closure. It can use the lisp programming language in order to satisfy this closure property.

   The closure property refers to ability of express the idea that complex things can be built using simple things. It is possible to generate very complex patterns by the different procedures that act on the painter.

Higher order operations can be achieved with procedure generators. The power lies in lisp's ability to create entirely new languages.

I am able to use the picture language and test it out using DrRacket and the SCIP package.

The lecture talks about the closure property. From I can follow I only need to implement some very basic primitives in order to implement the full stack of the picture language.

Frames are a definition of rectangles/canvas that are painter. A painter is an object that when painted draws a picture.

```
;; Allows for creation of a new procedure that represents a linear transformation
(define (frame-coord-map frame)
  (lambda (v)
    (add-vec (origin-frame frame)
             (add-vec (scale-vec (vecx v)
                                 (edge1-frame frame))
                      (scale-vec (vecy v)
                                 (edge2-frame frame))))))

;; takes list of segments and create a painter that draws line in those represente
(define (segments->painter segment-list)
  (lambda (frame)
    (for-each (lambda (segment)
                (draw-line ((frame-coord-map frame) (start-segment segment))
                           ((frame-coord-map frame) (end-segment segment))))
              segment-list)))
```

Using these functions it is possible to define new ways of creating painter objects in terms of other painter objects.

```
;; This will create a new painter that will flip the image upside down
(define (flip-vert painter)
  (transform-painter painter
                     (make-vect 0.0 1.0)
                     (make-vect 1.0 1.0)
                     (make-vect 0.0 0.0)))
```

```
;; self explanatory
(define (shrink-to-upper-right painter)
  (transform-painter painter
                     (make-vect 0.5 0.5)
                     (make-vect 1.0 0.5)
                     (make-vect 0.5 1.0)))

(define (rotate90 painter)
  (transform-painter painter
                     (make-vect 1.0 0.0)
                     (make-vect 1.0 1.0)
                     (make-vect 0.0 0.0)))

(define (squash-invards painter)
  (transform-painter painter
                     (make-vect 0.0 0.0)
                     (make-vect 0.65 0.35)
                     (make-vect 0.35 0.65)))
```

And now the all important beside function.

```
(define (beside painter1 painter2)
  (let ((split-point (make-vect 0.5 0.0)))
    (let ((paint-left (transform-painter painter1
                                         (make-vect 0.0 0.0)
                                         split-point
                                         (make-vect 0.0 1.0)))
          (paint-right (transform-painter painter2
                                          split-point
                                          (make-vect 1.0 0.0)
                                          (make-vect 0.5 1.0))))
      (lambda (frame)
        (paint-left frame)
        (paint-right frame)))))
```

All of this satisfies the closure property. The closure property seems to be an ability for lower level primitives

This idea of closure property allows for a `stratified` design in which one level solely depends on the lower levels. All computer science is based off of layers of abstraction. Lisp allows for language levels to be designed and use based one simple primitives the are provides on lower levels.

In theory a change in design or implementation should not have a significant effect on the upper layers of the language. There are many examples of this, but the picture language is the example given by the book.

Also the last exercise is skipped.

3 2.3 Symbolic Data

Scheme allows for the use of arbitrary symbols in as a means to work with data.

3.0.1 2.3.1 Quotation

Allows for expressing literal symbols inside of a an expression and not the values the the symbols themselves are supposed to represent. This can be accomplished with the quotation operator.

```
(define symbol 'define)
(define list-of-symbols '(a b c d))
;; quote function is possible. This allows for a keeping of standards
(define symbol2 (quote 'display))
(define other-list-of-symbols (quote (1 2 3 4)))
```

Allows for a differentiation between expressions to be evaluated and data representation that can be manipulated and explored. The `eq?` function allows for testing for the equality of symbols.

```
(define s 'a)
(eq? 'a s)
```

An implementation of `memq` is possible which checks a list for the presence of a specific symbol inside of it. If it is not contained then the function returns false. If it is contained then the function returns a sublist which the item as first entry.

```
(define (memq item x)
  (cond ((null? x) #f)
        ((eq? item (car x) )x)
        (else (memq item (cdr x)))))

(memq 'a (list  '0 'banana 'a 'b 'c))
```

3.0.2 2.3.2 Example: Symbolic Differentiation

One main goals of symbol manipulation using a computer language was the quest for algebraic manipulation by a computer. To be more especific the question to find a a way to find the derivative of a function and symbolic differentiation. This started the development of systems used by physicists and mathmathecians. The book will now begin to describe the thought proccess in creating a system for symbol manipulation.

1. Differentiation with Abstract Data

   The SICP implementation defines a set of differential properties that are kept in mind when beginning to implement a way to find derivative expressions.

   - $\frac{dc}{dx} = 0$ for any value
   - $\frac{dx}{dx} = 1$ identity
   - $\frac{d(u+v)}{dx} = \frac{du}{dv} + \frac{dv}{dx}$
   - $\frac{d(uv)}{dx} = u\frac{dv}{dx} + v\frac{du}{dx}$