fontfamily: libertinus

Notes on Abelson and Sussman, *Structure and Interpretation of Computer Programs*; page numbers reference the PDF version of the text available here

Chapter 2, "Building Abstractions with Data"

Cameron Clarke

August 2019

---

2. Building Abstractions with Data (pp. 107-293)

- A note on higher-order procedures, referencing chapter 1 (107-8)

  > We also saw that higher-order procedures enhance the power of our language by enabling us to manipulate, **and thereby to reason in terms of**, general methods of computation.

- The focus of chapter 2 is on **data that's more complex than the simple numerical data we used in chapter 1** (108)
- Comparative point about the focus of chapter 2 (vs. the focus of chapter 1) (108)

  > Thus, whereas our focus in chapter 1 was on building abstractions by combining procedures to form compound procedures, **we turn in this chapter to another key aspect of any programming language: the means it provides for building abstractions by combining data objects to form *compound data***.

- Why compound data are desirable in a programming language (108)

  > Why do we want compound data in a programming language? For the same reasons that we want compound procedures: **to elevate the conceptual level at which we can design our programs, to increase the modularity of our designs, and to enhance the expressive power of our language**. Just as the ability to define procedures enables us to deal with processes at a higher conceptual level than that of the primitive operations of the language, **the ability to construct compound data objects enables us to deal with data at a higher conceptual level than that of the primitive data objects of the language**.

- A note about designing ***compound data objects*** that are consistent with how we conceptualize the things in the world that these objects are meant

to represent (e.g., pairing together a numerator and a denominator as a compound data object to represent rational numbers) (109)

- A note about how compound data facilitate the design methodology of ***data abstraction*** (109)

  > The use of compound data also enables us to increase the modularity of our programs. If we can manipulate rational numbers directly as objects in their own right, then we can separate the part of our program that deals with rational numbers per se from the details of how rational numbers may be represented as pairs of integers. **The general technique of isolating the parts of a program that deal with how data objects are represented from the parts of a program that deal with how data objects are used is a powerful design methodology called *data abstraction*.**

- The example of a `linear-combination` procedure is given to illustrate how expressive power of the language increases when we allow for data abstraction through compound data (109-10)
  - The example also shows why...

    > [...] it is important that our programming language provide the ability to manipulate compound objects directly: **Without this, there is no way for a procedure such as `linear-combination` to pass its arguments along to `add` and `mul` without having to know their detailed structure**.

- Outline of the chapter (111-2)
  - Implement the rational-number arithmetic system given above as an example, as a way into the discussion of compound data and data abstractoion
    - A main point of discussion here is **abstraction as a technique for coping with complexity**
      - Will also talk about establishing **abstraction barriers** between different parts of a program
  - A discussion of how forming compound data requires that the programming language have some kind of "glue" so that data objects can be combined to form more complex data objects
    - In Scheme, **procedures can play the role of this "glue", so that no special "data" operations are required**
      - (A point about how this further blurs the distinction between "procedure" and "data")

    - A discussion here about ***closure***—"that the glue we use for combining data objects should allow us to combine not only primitive data objects, but compound data objects as well" (111)
    - A discussion about how compound data objects can serve as ***conventional interfaces*** for combining program modules in mix-and-match ways
  - A discussion about ***symbolic expressions***—"data whose elementary parts can be arbitrary symbols rather than only numbers" (111)
    - A discussion on how the time and space requirements of processes that manipulate data can depend on how the data themselves are represented
  - A discussion about ***generic operations*** which can handle many different types of data, whose implementation arises from the need of working with data that may be represented differently by different parts of the program
    - A discussion of maintaining modularity in the presence of generic expressions using ***data-directed programming*** "as a technique that allows individual data representations to be designed in isolation and then combined *additively* (i.e., without modification)" (112)

2.1 Introduction to Data Abstraction (pp. 112-132)

- ***data abstraction*** (112-3)

  > Data abstraction is a methodology that **enables us to isolate how a compound data object is used from the details of how it is constructed from more primitive data objects**.
  >
  > **The basic idea of data abstraction is to structure the programs that are to use compound data objects so that they operate on "abstract data."** That is, our programs should use data in such a way as to make no assumptions about the data that are not strictly necessary for performing the task at hand. At the same time, a "concrete" data representation is defined independent of the programs that use the data. The interface between these two parts of our system will be a set of procedures, called ***selectors*** **and** ***constructors***, that implement the abstract data in terms of the concrete representation.

2.1.1 Example: Arithmetic Operations for Rational Numbers (113-118)

- The note about ***wishful thinking*** as a powerful strategy of synthesis (in the context of constructors and selectors for rational numbers)

Pairs (115-6)

- Scheme provides a compound structure called a ***pair*, which is constructed using the primitive procedure `cons`** (115)

  > This procedure [`cons`] takes two arguments and returns a compound data object that contains the two arguments as parts.

  - `cons` stands for "construct" (115n2)
  - Can extract the parts of a given pair using the **primitive procedures `car` and `cdr`**
    - `car` stands for "Contents of Address part of Register" (115n2)
    - `cdr` (pronounced "could-er") stands for "Contents of Decrement part of Register" (115n2)
    - e.g., (115)

      ```
      (define x (cons 1 2))

      (car x)

      ;Value: 1

      (cdr x)

      ;Value: 2
      ```

  - `cons` can be used to define pairs whose elements are pairs, etc.

- Pairs (given that we can combine them) end up being a powerful, general-purpose building block to create many kinds of complex data structures (116)
  - Moreover, (116)

    > The single compound-data primitive *pair*, implemented by the procedures `cons`, `car`, and `cdr`, is the only glue we need.

  - Data objects constructed from pairs are called ***list-structured* data**

Representing rational numbers (116-8)

- Pairs are a natural way to represent the data associated with rational numbers (119)
  - The implementation that the authors give of `make-rat`, `numer`, and `denom` are then in terms of `cons`, `car`, and `cdr`, respectively

Exercises (118)

- Exercise 2.1
  - Define the alternative `make-rat` as follows:

```
(define (make-rat a b)
  (cond ((positive? (* a b)) (cons (abs a) (abs b))) ; this is the most parsimoniou
        (else (cons (-(abs a)) (abs b)))))
```

2.1.2 Abstraction Barriers (118-122)

- Underlying idea of data abstraction (118)

  > In general, **the underlying idea of data abstraction is to identify for each type of data object a basic set of operations in terms of which all manipulations of data objects of that type will be expressed, *and then to use only those operations in manipulating the data*** [emphasis added].

- ***abstraction barriers*** isolate different "levels" of the system (119)

  > At each level, the barrier separates the programs (above) that use the data abstraction from the programs (below) that implement the data abstraction.

  - Maintaining abstraction barriers helps users not have to think about **implementation details** of lower levels of abstraction
  - Another note (121)

    > Constraining the dependence on the representation to a few interface procedures helps us design programs as well as modify them, because it allows us to maintain the flexibility to consider alternate implementations.

Exercise (121-2)

- Exercise 2.2
  - We'll have the following definitions:

```
(define (make-segment point-a point-b)
  (cons point-a point-b))

(define (start-segment segment)
  (car segment))

(define (end-segment segment)
  (cdr segment))
```

```
(define (make-point x y)
  (cons x y))

(define (x-point point)
  (car point))

(define (y-point point)
  (cdr point))

(define (midpoint-segment segment)
  (let ((x1 (x-point (start-segment segment)))
        (y1 (y-point (start-segment segment)))
        (x2 (x-point (end-segment segment)))
        (y2 (y-point (end-segment segment)))
        (avg (lambda (a b) (/ (+ a b) 2.0))))
       (make-point (avg x1 x2)
                   (avg y1 y2))))
```

And define the `print-point` procedure presented in the text on pg. 121-2

```
(define (print-point p)
  (newline)
  (display "(")
  (display (x-point p))
  (display ",")
  (display (y-point p))
  (display ")"))
```

And now let's test out what we've implemented:

```
(define a (make-point 1 2))
;Value: a

(x-point a)
;Value: 1

(y-point a)
;Value: 2

(define b (make-point 3 4))
;Value: b

(x-point b)
;Value: 3

(y-point b)
;Value: 4
```

```
(print-point a)
(1,2)
;Unspecified return value

(print-point b)
(3,4)
;Unspecified return value

(define seg (make-segment a b))
;Value: seg

(start-segment seg)
;Value: (1 . 2)

(end-segment seg)
;Value: (3 . 4)

(define midpoint (midpoint-segment seg))
;Value: midpoint

(print-point midpoint)
(2.,3.)
;Unspecified return value
```

- Exercise 2.3
  - See exercise_2_3.txt for one implementation. TODO: Implement a second way.

2.1.3 What is Meant by Data? (122-126)

- A perspective from the authors (123)

  > In general, we can think of data as defined by some collection of selectors and constructors, together wish specified conditions that these procedures must fulfill in order to be a valid representation.

  - (123n5 has some links to literature that might be interesting to follow up on
- The example of the alternative implementation of `cons`, `car`, and `cdr` using only procedures and no "real" data structures (124)
  - All this is given an exact specification of the following desideratum about `cons`, `car`, and `cdr` and how they relate to one another: (123)

    > [...] for any objects `x` and `y`, if `z` is `(cons x y)` then `(car z)` is `x` and `(cdr z)` is `y`.

- A general word about the procedural implementation above: ***message passing*** (125)

  > This example also demonstrates that the ability to manipulate procedures as objects automatically provides the ability to represent compound data. This may seem a curiosity now, but procedural representations of data will play a central role in our programming repertoire. **This style of programming is often called *message passing*, and we will be using it as a basic tool in Chapter 3 when we address the issues of modeling and simulation.**

Exercises (125-6)

- Exercise 2.4
  - Consider the evaluation of

    ```
    (car (cons x y))
    ```

    using the substitution model:

    ```
    (car (cons x y))

    ((cons x y) (lambda (p q) p))

    ((lambda (m) (m x y)) (lambda (p q) p))

    ((lambda (p q) p) x y)

    x
    ```

    Now consider a definition of a `cdr` procedure as follows:

    ```
    (define (cdr z)
      (z (lambda (p q) q)))
    ```

    The expression `(cdr (cons x y))` would evolve as follows:

    ```
    (cdr (cons x y))

    ((cons x y) (lambda (p q) q))

    ((lambda (m) (m x y)) (lambda (p q) q))

    ((lambda (p q) q) x y)

    y
    ```

- Exercise 2.5

- Define `cons`:

```
(define (cons a b)
  (* (expt 2 a) (expt 3 b)))
```

Now define a helper procedure `find-prime-factor-exponent`:

```
(define (find-prime-factor-exponent n prime-factor)
  (if (not (= (remainder n prime-factor) 0))
      0
      (+ 1 (find-prime-factor-exponent (/ n prime-factor) prime-factor))))
```

Define `car`:

```
(define (car x)
  (find-prime-factor-exponent x 2))
```

Define `cdr`:

```
(define (cdr x)
  (find-prime-factor-exponent x 3))
```

Test it out:

```
(car (cons 0 0 ))
;Value: 0

(cdr (cons 0 0))
;Value: 0

(car (cons 1 2))
;Value: 1

(cdr (cons 1 2))
;Value: 2

(car (cons 30 72))
;Value: 30

(cdr (cons 30 72))
;Value: 72
```

- Exercise 2.6
  - It seems like we need to be careful about not conflating/confusing the names of various similarly-named formal parameters in the procedures and `lambda` special forms at play. With this in mind let's work out what `one` would be, given that it's equivalent to `(add-1 zero)`:

```
(add-1 zero)
(add-1 (lambda (f) (lambda (x) x)))
(lambda (f2) (lambda (x2) (f2 (((lambda (f1) (lambda (x1) x1)) f2) x2))))
```

```
(lambda (f2) (lambda (x2) (f2 ((lambda (x1) x1) x2))))
(lambda (f2) (lambda (x2) (f2 x2)))
```

So we define `one` as follows:

```
(define one (lambda (f) (lambda (x) (f x))))
```

And now let's derive a representation for `two` by considering `(add-1 one)`:

```
(add-1 one)
(add-1 (lambda (f) (lambda (x) (f x))))
((lambda (f) (lambda (x) (f ((n f) x)))) (lambda (f) (lambda (x) (f x))))
(lambda (f2) (lambda (x2) (f2 (((lambda (f1) (lambda (x1) (f1 x1))) f2) x2))))
(lambda (f2) (lambda (x2) (f2 (f2 x2))))
```

So we can define `two` as follows:

```
(define two (lambda (f) (lambda (x) (f (f x)))))
```

Let's now define our `plus` procedure (we call it `plus` instead of `+`):

```
(define (plus n m)
  (lambda (f) (lambda (x) ((m f) ((n f) x)))))
```

Now let's look at some examples of one particular application of the procedures we've defined, that make use of numerals and a procedure called `succ` that implements a numeric successor function (and makes use of ordinary numerical addition with the built-in `+` operator):

```
(define (succ x) (+ x 1))
```

Note first that we can recover some ordinary numeric meaning from the Church numerals we've implemented as follows:

```
((zero succ) 0)
;Value: 0

((zero succ) 1)
;Value: 1

((one succ) 0)
;Value: 1

((two succ) 0)
;Value: 2

(((add-1 one) succ) 0)
;Value: 2

(((add-1 two) succ) 0)
;Value: 3
```

Let's now look at how our `plus` procedure functions in this context:

```
(((plus zero zero) succ) 0)
;Value: 0

(((plus zero one) succ) 0)
;Value: 1

(((plus zero two) succ) 0)
;Value: 2

(((plus one two) succ) 0)
;Value: 3

(((plus two two) succ) 0)
;Value: 4

(((plus two two) succ) 1)
;Value: 5

(((plus two two) succ) 2)
;Value: 6

(((plus (add-1 two) (add-1 zero)) succ) 3)
;Value: 7

(((plus (add-1 two) (add-1 (add-1 zero))) succ) 3)
;Value: 8

(((plus (add-1 two) (add-1 (add-1 zero))) succ) 4)
;Value: 9
```

2.1.4 Extended Exercise: Interval Arithmetic (126-132)

Exercises (128-32)

- Exercise 2.7
  - Define our selectors:

```
(define (lower-bound interval)
  (min (car interval) (cdr interval)))

(define (upper-bound interval)
  (max (car interval) (cdr interval)))
```

- Exercise 2.8

  - Define an interval subtraction procedure called `sub-interval`:

    ```
    (define (sub-interval x y)
      (make-interval (- (lower-bound x) (lower-bound y))
                     (- (upper-bound x) (upper-bound y))))
    ```

    (Alternatively, we could define `sub-interval` in terms of `add-interval`.)

- Exercise 2.9
  - [TODO]
- Exercise 2.10
  - [TODO]
- Exercise 2.11
  - [TODO]
- Exercise 2.12
  - [TODO]
- Exercise 2.13
  - [TODO]
- Exercise 2.14
  - [TODO]
- Exercise 2.15
  - [TODO]
- Exercise 2.16
  - [TODO]

2.2 Hierarchical Data and the Closure Property (pp. 132-192)

2.3 Symbolic Data (pp. 192-229)

2.4 Multiple Representations for Abstract Data (pp. 229-254)

2.5 Systems with Generic Operations (pp. 254-293)