Notes on Abelson and Sussman, *Structure and Interpretation of Computer Programs*

Chapter 1, "Building Abstractions with Procedures"

Cameron Clarke

July 2019

---

1. Building Abstractions with Procedures (pp. 1-106)

- Target of study: **computational processes** (1)
  - intuition: "Computational processes are abstract beings that inhabit computers" (1)
- Processes manipulate **data** (1)
- Evolution of a process is directed by a pattern of rules called a **program** (2)
- An upshot: "**People create programs to direct processes**" (2)
- Programs are composed from **programming languages** (2)
- A framing thought, among others (2-3)

  > Well-designed computational systems, like well-designed automobiles or nuclear reactors, are designed in a modular manner, so that parts can be constructed, replaced, and debugged separately.

Programming in Lisp

- "[. . .] our **procedural** thoughts will be expressed in Lisp" (3)
- Lisp **interpreter**: "a machine that carries out the processes described in the Lisp language" (3)
- Interesting historical note, about one of Lisp's original use cases (4)

  > Lisp [. . .] was designed to provide symbol-manipulating capabilities for attacking programming problems such as the symbolic differentiation and integration of algebraic expressions.

- A note about dialects of Lisp (4)

  > Lisp is by now a family of dialects, which, while sharing most of the original features, may differ from one another in significant ways

  - This book makes use of the Scheme dialect of Lisp
- A main reason why Lisp is being used as the framework for the book's discussion of programming (5)

> If Lisp is not a mainstream language, why are we using it as the framework for our discussion of programming? Because the language possesses unique features that make it an excellent medium for studying important programming constructs and data structures for relating them to the linguistic features that support them. **The most significant of these features is the fact that Lisp descriptions of processes, called *procedures*, can themselves be represented and manipulated as Lisp data**. The importance of this is that there are powerful program-design techniques that rely on the ability to **blur the traditional distinction between "passive" data and "active" processes**.

- The above facts make Lisp good at writing programs that **manipulate other programs as data** (like interpreters and compilers)

1.1 The Elements of Programming (pp. 6-39)

- A powerful programming language can be used as: (6)
  - a means for instructing a computer to perform tasks
  - a framework within which we organize our ideas about processes
- Combining simple ideas into complex ideas (6)

  > [...] when we describe a language, we should **pay particular attention to the means that the language provides for combining simple ideas to form more complex ideas**.

  - Three mechanisms for doing this
    1. **Primitive expressions**, which represent the simplest entities the language is concerned with
    2. **Means of combination**, by which compound elements are built from simpler ones
    3. **Means of abstraction**, by which compound elements can be named and manipulated as units
- Two kinds of elements in programming: **procedures** and **data** (6)
  - Data is the "stuff" that we manipulate; procedures are the rules for manipulating the data
- An upshot: (6)

  > [...] any powerful programming language should be able to describe primitive data and primitive procedures and should have methods for combining and abstracting procedures and data.

1.1.1 Expressions (pp. 7-10)

- An initial framing (7)

  > Imagine that you are sitting at a computer terminal. You type an *expression*, and the interpreter responds by displaying the result of its *evaluating* that expression.

- Some vocab (8)
  - **combinations**: expressions formed by delimiting a list of expressions within parentheses, in order to denote procedure application
  - **operator**: leftmost element in the list
  - **operand**: other elements in the list than the operator
  - **argument**: "The value of a combination is obtained by applying the procedure specified by the operator to the *arguments* that are the values of the operands"
- Scheme uses **prefix notation**: operator to the left of the operands (8)
  - Advantages of prefix notation (8-9)
    - can accommodate procedures that may take an arbitrary number of arguments
      - e.g., `(+ 21 35 12 7 5 6)`, which evaluates to `86`
    - extends in a straightforward way to allow combinations to be *nested*, i.e., combinations themselves can have combinations as elements
      - e.g., `(+ (* 3 5) (- 10 6))`, which evaluates to `19`
      - might think to use a *pretty-printing* convention—where each long combination is written so that the operands are aligned vertically—if you have lots of nested things, like

        ```
        (+ (* 3
              (+ (* 2 4)
                 (+ 3 5)))
           (+ (- 10 7)
              6))
        ```

        instead of

        ```
        (+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))
        ```

- Scheme interpreter runs in a **REPL** (*read-eval-print loop*)
  - Interesting note here (10)

    > Observe in particular that it is not necessary to explicitly instruct the interpreter to print the value of the expression.

1.1.2 Naming and the Environment (pp. 10-12)

- **names** to refer to computation objects (10)

  > We say that the name identifies a *variable* whose *value* is the object.

- The Scheme dialect of Lisp uses `define` to name things (10)

  - e.g.,

```
(define size 2)
```

- `define` is Scheme's **simplest means of abstraction**, since it lets us to use simple names to refer to the results of compound operations (11)

- An upshot (11)

  > [...] complex programs are constructed by building, step by step, computational objects of increasing complexity.

- In order to associate values with symbols, need to have some sort of memory that keeps track of the name object pairs (11)

  - This memory is called the **environment** (more precisely, the **global environment** in this specific case)

1.1.3 Evaluating Combinations (pp. 12-15)

- The interpreter itself is following a **procedure** when it evaluates combinations (12)

  > To evaluate a combination, do the following:
  >
  > 1. Evaluate the subexpression of the combination.
  > 2. Apply the procedure that is the value of the leftmost subexpression (the operator) to the arguments that are the values of the other subexpressions (the operands).

- Important points about processes in general, that are illustrated by the above example (12-14)

  - The evaluation rule is **recursive**: the first step above tells us to *recursively* evaluate the evaluation process on each element of a given combination (12)

    * Can think of this recursive evaluation process as a process of **tree accumulation** (where the tree structure is implied by the nested structure of the combinations)—the **values of the operands "percolate upward"**

  - repeated application of the first step in the evaluation rule brings us to the point where we need to evaluate **primitive expressions** (14)

    * examples of primitive expressions are numerals, built-in operators, and other names
    * Rules for handling primitive cases (14)

      > We take care of the primitive cases by stipulating that
      >
      > · the values of numerals are the numbers that they name,
      > · the values of built-in operators are the machine instruction sequences that carry out the corresponding operations, and
      > · the values of other names are the objects associate with those names in the environment.

    * A note about the relationship between meaning of symbols in expressions and the *environment* (14)

      > We may regard the second rule [from the rules for handling primitive cases, above] as a special case of the third one by stipulating that symbols such as + and `*` are also included in the global environment, and are associated with the sequences of machine instructions that are their "values." **The key point to notice is the role of the environment in determining the meaning of the symbols in expressions**. In an interactive programming language such as Lisp, it is meaningless to speak of the value of an expression such as `(+ x 1)` without specifying any information about the environment that would provide a meaning for the symbol x (or even for the symbol +).

  - The evaluation rule does not handle definitions (14)
    * Corollary: `(define x 3)`, for example, is not a combination

- Exceptions to the general evaluation rule are called **special forms** (14)
  - Each special form has its own evaluation rule
- Note about **syntax** of the programming language (14-5)

  > The various kinds of expressions (each with its associated evaluation rule) constitute the syntax of the programming language.

- Lisp's syntax is simple relative to other programming languages' syntaxes (15)

  > [In Lisp], the evaluation rule for expressions can be described by as simple general rule together with specialized rules for a small number of special forms.

- A note here also about ***syntactic sugar*** (15n11)

  > Special syntactic forms that are simply convenient alternative surface structures for things that can be written in more uniform ways are

sometimes called *syntactic sugar*, to use a phrase coined by Peter Landin.

1.1.4 Compound Procedures (pp. 15-18)

- Elements that Lisp has, that must appear in any "powerful" programming language (15)
    - Numbers and arithmetic operations are primitive data and procedures
    - Nesting of combinations provides a means of combining operations
    - Definitions that associate names with values provide a limited means of abstraction
- This section focuses on **procedure definitions**, which are… (15)

    > [...] a much more powerful abstraction technique by which a **compound operation can be given a name and then referred to as a unit**.

- Example: the "squaring" operation (16)

```
(define (square x) (* x x))
```

- Could read/understand this as: "To (`define`) square (`square`) something (x), multiply (`*`) it (x) by itself (x)"
    - Interesting description (16)

        > The thing to be multiplied is given a local name, `x`, which plays the same role that a pronoun plays in natural language.

- This is an example of a **compound procedure** (16)

    > Evaluating the definition creates this compound procedure and associates it with the name `square`.

- General form of a procedure definition (16-7)

```
(define ([name] [formal parameters]) [body])
```

- `[name]` is a symbol to be associated with the procedure definition in the environment
- `[formal parameters]` are the names used within the body of the procedure to refer to the corresponding arguments of the procedure
- `[body]` is an expression that will yield the value of the procedure application when the formal parameters are replaced by the actual arguments to which the procedure is applied

    * `[body]` could also include names of procedures that have already been defined—e.g., we could use `square`, which we defined above, as a building block to define a `sum-of-squares` procedure (this example is given in the book)

    * In general, `[body]` can be a sequence of expressions (17n14)

1.1.5 The Substitution Model for Procedure Application (pp. 18-22)

- Evaluating a combination whose operator names a *compound* procedure is similar to evaluating a combination whose operator is a *primitive* procedure (18)

  > That is, the interpreter evaluates the elements of the combination and applies the procedure (which is the value of the operator of the combination) to the arguments (which are the values of the operands of the combination).

- **Application process for compound procedures** (18)

  > To apply a compound procedure to arguments, evaluate the body of the procedure with each formal parameter replaced by the corresponding argument.

  - (We assume that the mechanism for applying primitive procedures to arguments is built into the interpreter)

- **substitution model** for procedure application (18-9)

  - An overview from the example given in the book (19)

    > Evaluating this combination [the one in the example in the book] involves three subproblems. We must evaluate the operator to get the procedure to be applied, and we must evaluate the operands to get the arguments.

  - The substitution model can be thought of as a model that determines the "meaning" of procedure application—but it's **not the whole (or accurate) story**; it's just a helpful way to think about procedure application (19)

    * **Two points** to stress here (19-20)

      1. Typical interpreters do not actually evaluate procedure applications by manipulating the text of a procedure to substitute values for formal parameters; in practice this happens using a **local environment** for the formal parameters

      2. The substitution model is relatively simple, but it doesn't correctly model all that we'll eventually need to model; in particular, the substitution model breaks down when we try to use procedures with "mutable data" (this will be addressed in Chapter 3)

Applicative order versus normal order (20-22)

- The intuition behind **normal-order evaluation**: **"fully expand and then reduce"** (21)
  - i.e., first substitute operand expressions for parameters until there are only primitive operators, and then perform evaluation
- This stands in contrast to the **applicative-order evaluation**, which says: **"evaluate the arguments and then apply"** (21)
  - This is the kind of evaluation we've discussed in previous sections
- **Lisp uses applicative-order evaluation** (21)
- A claim about an equivalence result for normal-order evaluation and applicative-order evaluation, under certain conditions (21)

  > It can be shown that, for procedure applications that can be modeled using substitution (including all the procedures in the first two chapters of this book) and that yield legitimate values, **normal-order and applicative-order evaluation produce the same value**.

- Reasons why Lisp uses applicative-order evaluation (21-2)
  - Partly because of the additional efficiency obtained from avoiding multiple evaluations of expressions
  - More significantly—normal-order evaluation becomes much more complicated to deal with when considering procedures that can't be modeled using substitution
  - (Even given these reasons, normal-order evaluation can be a valuable tool. More on this in Chapters 3 and 4.)

1.1.6 Conditional Expressions and Predicates (pp. 22-28)

- The cond construct (short for "conditional") is used for doing **case analysis** in Lisp (22)
  - e.g.,

    ```
    (define (abs x)
      (cond ((> x 0) x)
            ((= x 0) 0)
            ((< x 0) (- x))))
    ```

  - general form:

    ```
    (cond ([p_1] [e_1])
          ([p_2] [e_2])
          ([p_3] [e_3])
    ```

```
    ...
    ([p_n] [e_n]))
```

- each of the `([p_i] [e_i])` are called **clauses**; the `[p_i]` are **predicates**, and the `[e_i]` are **consequent expressions** (22)
- In conditional expressions, the interpreter (informally put) "goes until it finds a predicate whose value is true"; at this point the interpreter returns the value of the corresponding consequent expression as the value of the conditional expression overall
  - In Lisp, the constant `#t` denotes "true" and the constant `#f` denotes "false", in a sense (23n17)
- If none of the `[p_i]` is found to be true, the value of the `cond` is undefined

- The `else` symbol can be used in place of the predicate in the final clause of `cond` (24)
  - Example: could've also defined `abs` above as

    ```
    (define (abs x)
      (cond ((< x 0) (- x))
            (else x)))
    ```

  - In fact, can use any expression that always evaluates to a true value like `else` here, in place of the final predicate

- Can also use `if`, an example of a **special form**, to define the absolute-value procedure (24)

```
(define (abs x)
  (if (< x 0)
      (- x)
      x))
```

  - In general: (24)

    ```
    (if [predicate] [consequent] [alternative])
    ```

- There are **logical composition operations**, which allow us to construct **compound predicates** (24-5)
  - and: `(and [e_1] ... [e_n])`

    > The interpreter evaluates the expressions [`[e]`] one at a time, in left-to-right order. **If any [`[e]`] evaluates to false, the value of the `and` expression is false and the rest of the [`[e]`]'s** [sic] **are not evaluated. If all [`[e]`]'s** [sic] **evaluate to true values, the value of the end expression is the value of the last one.**

  - or: `(or [e_1] ... [e_n])`

The interpreter evaluates the expressions `[[e]]` one at a time, in left-to-right order. **If any `[[e]]` evaluates to a true value, that value is returned as the value of the `or` expression, and the rest of the `[[e]]`'s** [sic] **are not evaluated. If all `[[e]]`'s** [sic] **evaluate to false, the value of the `or` expression is false.**

  - `not`: `(not [e])`

    The value of a `not` expression is true when the expression `[[e]]` evaluates to false, and false otherwise.

- Important: **`and` and `or` are special forms, not procedures** (25)
  - This is because the subexpressions are not necessarily all evaluated with `and` and `or`
  - (`not` is an ordinary procedure, though)

Exercises (26-8)

- Exercise 1.1
  1. `10`

     prints

     ;Value: 10
  2. `(+ 5 3 4)`

     prints

     ;Value: 12
  3. `(- 9 1)`

     prints

     ;Value: 8
  4. `(/ 6 2)`

     prints

     ;Value: 3
  5. `(+ (* 2 4) (- 4 6))`

     prints

     ;Value: 6
  6. `(define a 3)`

     prints

     ;Value: a
  7. `(define b (+ a 1))`

prints

```
;Value: b
```

8. `(+ a b (* a b))`

   prints

   ```
   ;Value: 19
   ```

9. `(= a b)`

   prints

   ```
   ;Value: #f
   ```

10. ```
    (if (and (> b a) (< b (* a b)))
        b
        a)
    ```

    prints

    ```
    ;Value: 4
    ```

11. ```
    (cond ((= a 4) 6)
          ((= b 4) (+ 6 7 a))
          (else 25))
    ```

    prints

    ```
    ;Value: 16
    ```

12. `(+ 2 (if (> b a) b a))`

    prints

    ```
    ;Value: 6
    ```

13. ```
    (* (cond ((> a b) a)
             ((< a b) b)
             (else -1))
       (+ a 1))
    ```

    prints

    ```
    ;Value: 16
    ```

- Exercise 1.2

  - ```
    (/ (+ 5
          4
          (- 2
             (- 3
                (+ 6
                   (/ 4 5)))))
       (* 3
    ```

```
    (- 6 2)
    (- 2 7)))
```

- Exercise 1.3
  - Making use of the `square` and `sum-of-squares` procedures defined earlier in the chapter:

    ```
    (define (square x) (* x x))

    (define (sum-of-squares x y)
      (+ (square x) (square y)))
    ```

    We can define the desired procedure in a few ways.

    First, using `cond` and tests for strict inequality:

    ```
    (define (ssq-largest-two-cond a b c)
            (cond ((and (> a b) (> b c)) (sum-of-squares a b))
                  ((and (> a b) (> c b)) (sum-of-squares a c))
                  ((> a c)               (sum-of-squares a b)) ;implicitly, (> b a)
                  (else                   (sum-of-squares b c))))
    ```

    Could also implement this using `if`:

    ```
    (define (ssq-largest-two-if a b c)
            (if (> a b)
                (if (> b c)
                    (sum-of-squares a b)
                    (sum-of-squares a c))
                (if (> a c)
                    (sum-of-squares a b)
                    (sum-of-squares b c))))
    ```

    And finally, making use of the `>=` operator allows us to simplify our case analysis a bit (this solution is based on the one given here):

    ```
    (define (ssq-largest-two-cond-geq a b c)
            (cond ((and (>= a c) (>= b c)) (sum-of-squares a b))
                  ((and (>= a b) (>= c b)) (sum-of-squares a c))
                  ((and (>= b a) (>= c a)) (sum-of-squares b c))))
    ```

- Exercise 1.4
  - If the value of `b` is greater than `0`, then the combination `(a-plus-abs-b a b)` will evaluate to the sum of `a` and `b` (i.e., `(+ a b)`). Else, the combination `(a-plus-abs-b a b)` will evaluate to the difference of `a` and `b` (i.e., `(- a b)`).
- Exercise 1.5
  - Under normal-order evaluation, the value of the expression `(test 0 (p))` will be `0`. Under applicative-order evaluation, the expression `(test 0`

`(p))` will trigger an infinite loop. The crucial difference is that under normal-order evaluation, *the expression `(p)` is never evaluated when x has value 0*, because operands are not evaluated until their values are needed—and here, since the predicate of the `if` evaluates to `#t` when the value of `x` is `0`, the `(p)` expression is never evaluated. But, under applicative-order evaluation, `(p)` *is* evaluated, since all operators and operands are evaluated before the procedure is applied to any arguments.

* A crucial thing here that I got confused about the first time I went to answer this exercise—even though things are "fully expanded" under normal-order evaluation, this *does not mean* that each of the resulting expressions (which are all comprised of primitive expressions) is evaluated!

* So we might think that applicative-order evaluation and normal-order evaluation differ along two axes

  1. The axis of expansion; and

  2. The axis of application order

     · For normal-order, we might paraphrase as "expand early, apply just-in-time". For applicative-order, we might paraphrase as "evaluate early, apply all at once".

1.1.7 Example: Square Roots by Newton's Method (pp. 28-33)

- A difference between mathematical functions and computer procedures (28)

  > Procedures must be effective.

  – The book frames this distinction as (in part, at least) one of construction vs. recognition

  – Also, a distinction between *declarative* knowledge and *imperative* knowledge (28-9)

    > In mathematics, we are usually concerned with declarative (what is) descriptions, whereas in computer science we are usually concerned with imperative (how to) descriptions.

  – (A discussion of an implementation of Newton's method is given here)

    * An interesting note (31-2)

      > The `sqrt` program also illustrates that the simple procedural language we have introduced so far is sufficient for writing any purely numerical program that one could write in, say, C or Pascal. This might seem surprising, since we have not included in our language any iterative (looping) constructs that direct the computer to do something over and over again. `sqrt-iter`, on the other hand, demonstrates how iteration

can be accomplished using no special construct other than the ordinary ability to call a procedure.

· (The book includes a pointer here to Section 1.2.1, which discusses *tail recursion*)

Exercises (32-3)

- Exercise 1.6
  - Using the `new-if` implementation of `sqrt-iter` results in an **infinite loop** because of applicative-order evaluation—the alternative expression of `new-if` will be evaluated every single time `new-if` is applied, even if the predicate evaluates to true. This is distinct from the behavior of `if` where the alternative expression will `not` be evaluated if the predicate is true (thus allowing for proper termination of the procedure.)
- Exercise 1.7
  - With the following procedures as implemented in the book:

```
(define (average x y)
  (/ (+ x y) 2))

(define (square x) (* x x))

(define (improve guess x)
  (average guess (/ x guess)))

(define (good-enough? guess x)
  (< (abs (- (square guess) x)) 0.001))

(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x) x)))

(define (sqrt x)
  (sqrt-iter 1.0 x))
```

Consider what happens for a number less than `0.001`, for example:

```
(* (sqrt 0.0009) (sqrt 0.0009))
```

```
;Value: 1.6241401856992538e-3
```

Now consider what happens for a large number, e.g., `10000000000000000000000`

```
(* (sqrt 10000000000000000000000) (sqrt 10000000000000000000000))

; results in an infinite loop because changes to 'guess' are never acceptably small for 'go
```

An alternative implementation fo `good-enough?` tests how "close" (in a sense) `guess` is from `(improve guess x)`, for a given value of `guess`, and only updates `guess` if `(/ (- guess (improve guess x)) guess)` is large enough in magnitude. Call this alternative implementation `guesses-close-enough?`

```
(define (guesses-close-enough? guess x)
  (< (abs (/ (- guess (improve guess x)) guess)) 0.001))
```

And now define alternative implementations of `sqrt-iter` and `sqrt` that make use of `guesses-close-enough?`:

```
(define (sqrt-iter-alt guess x)
  (if (guesses-close-enough? guess x)
      guess
      (sqrt-iter-alt (improve guess x) x)))

(define (sqrt-alt x)
  (sqrt-iter-alt 1.0 x))
```

Let's see how `sqrt-alt` handles the case of a very small number:

```
(* (sqrt-alt 0.0009) (sqrt-alt 0.0009))

;Value: 9.016608107957646e-4
```

And now try our cases of large numbers:

```
(* (sqrt-alt 10000000000000000000000) (sqrt-alt 10000000000000000000000))

;Value: 1.0000036472100744e23
```

So it seems like `guesses-close-enough?` results in a square root procedure (`sqrt-alt`) that solves for the issues we were seeing before!

- Exercise 1.8
  - Follow a similar template as `sqrt-alt` from above:

```
(define (square x) (* x x))

(define (cube x) (* x x x))

(define (improve-cube-rt guess x)
   (/ (+ (/ x
            (square guess))
         (* 2 guess))
      3))
```

```scheme
(define (guesses-close-enough-cube-rt? guess x)
  (< (abs (/ (- guess (improve-cube-rt guess x)) guess)) 0.001))

(define (cube-rt-iter-alt guess x)
  (if (guesses-close-enough-cube-rt? guess x)
      guess
      (cube-rt-iter-alt (improve-cube-rt guess x) x)))

(define (cube-rt-alt x)
  (cube-rt-iter-alt 1.0 x))
```

Now let's check some values:

```scheme
(cube-rt-alt 0)
;Value: 4.9406564584124654e-324
```

```scheme
(cube-rt-alt 8)

;Value: 2.000004911675504
```

```scheme
(cube-rt-alt -27)

;Value: -3.0001270919925287
```

```scheme
(* (cube-rt-alt 100000000000000000000000)
   (cube-rt-alt 100000000000000000000000)
   (cube-rt-alt 100000000000000000000000))

;Value: 1.0000497722096368e23
```

```scheme
(* (cube-rt-alt 0.0009)
   (cube-rt-alt 0.0009)
   (cube-rt-alt 0.0009))

;Value: 9.006382994805101e-4
```

So it looks like our cube root procedure is working how we'd want, including for very large and very small numbers.

1.1.8 Procedures as Black-Box Abstractions (pp. 33-39)

- A note about `sqrt` (33)

  `sqrt` is our first example of a process defined by a set of **mutually defined procedures**.

- A note about the design decision behind how we decomposed the square root procedure (34)

> Observe that the problem of computing square roots breaks up naturally into a number of subproblems: how to tell whether a guess is good enough, how to improve a guess, and so on. [...] The importance of this decomposition strategy is not simply that one is dividing the program into parts. [...] **Rather, it is crucial that each procedure accomplishes an identifiable task that can be used as a module in defining other procedures**.

- The key thing here is ***procedural abstraction*** (34-5)

> For example, when we define the `good-enough?` procedure in terms of `square`, we are able to regard the `square` procedure as a "black box." We are not at that moment concerned with *how* the procedure computes its result, only with the fact that it computes the square. **The details of how the square is computed can be suppressed, to be considered at a later time. Indeed, as far as the `good-enough?` procedure is concerned, `square` is not quite a procedure but rather an abstraction of a procedure, a so-called *procedural abstraction*. At this level of abstraction, any procedure that computes the square is equally good.**

1.2 Procedures and the Processes They Generate (pp. 40-74)

1.3 Formulating Abstractions with Higher-Order Procedures (pp. 74-106)