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 (3-5)

- "[...] 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., `1000000000000000000000000`

```
(* (sqrt 100000000000000000000000) (sqrt 100000000000000000000000))

; 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 100000000000000000000000) (sqrt-alt 100000000000000000000000))

;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))
```

```
(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:

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

(cube-rt-alt 8)

;Value: 2.000004911675504

(cube-rt-alt -27)

;Value: -3.0001270919925287

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

;Value: 1.0000497722096368e23

(* (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.**

  - A heuristic (35)

    > So a procedure definition should be able to suppress detail. The users of the procedure may not have written the procedure themselves, but may have obtained it from another programmer as a black box. **A user should not need to know how the procedure is implemented in order to use it**.

Local names (35-7)

- Choice of names for the procedure's formal parameters is an implementation detail that should not matter to users (35)
  - A consequence: parameter names of a procedure **must be local to the body of the procedure** (36)
    - An illustration of this, using the `square` and `good-enough?` procedures from before (36)

      > [...] The argument of `square` is `guess`. If the author of square used x (as above) to refer to that argument, we see that the x in `good-enough?` must be a different x than the one in `square`. Running the procedure `square` must not affect the value of x that is used by `good-enough?`, because that value of x may be needed by `good-enough?` after `square` is done computing.
      >
      > If the parameters were not local to the bodies of their respective procedures, then the parameter x in `square` could be

> confused with the parameter `x` in `good-enough?`, and the behavior of `good-enough?` would depend upon which version of `square` we used. **Thus, `square` would not be the black box we desired**.

- On the importance of formal parameters of procedures (36-7)

  > A formal parameter of a procedure has a very special role in the procedure definition, in that **it doesn't matter what name the formal parameter has. Such a name is called a *bound variable*, and we say that the procedure definition *binds* its formal parameter.** The meaning of a procedure definition is unchanged if a bound variable is consistently renamed throughout the definition. **If a variable is not bound, we say that it is *free*. The set of expressions for which a binding defines a name is called the *scope* of that name. In a procedure definition, the bound variables declared as the formal parameters of the procedure have the body of the procedure as their scope**.

- In the example of `good-enough?` the variables `<`, `-`, and `abs` are free
  - A thought came up here: as the authors have described them, aren't "free" and "bound" *relative* notions? So like the variable `<` is free *relative to* `good-enough?` or something like this.

- The notion of **capturing** a variable (37)

  > [In the definition of `good-enough?`, if] we renamed `guess` to `abs` we would have introduced a bug by ***capturing*** the variable `abs`. It would have changed from free to bound.

- Just because a variable is free within a given procedure does not mean that the *meaning* of the free variable is irrelevant to the procedure, e.g., the meaning of `abs` certainly matters for the correctness of the `good-enough?` procedure (37)

Internal definitions and block structure (37-9)

- What was just described above is a kind of **name isolation**

- Consider another kind of name isolation, relating to potential implementations the square-root program: localized subprocedures using **block structure** (38)

  > The problem with this program [as it was implemented in earlier sections] is that the only procedure that is important to users of `sqrt` is `sqrt`. The other procedures (`sqrt-iter`, `good-enough?`, and `improve`) only clutter up their minds. [...] We would like to **localize the subprocedures, hiding them inside `sqrt` so that `sqrt` could coexist with other successive approximations, each having its own private `good-enough?` procedure. To make this**

possible we allow a procedure to have internal definitions that are local to that procedure.

- Can also leverage **lexical scoping** to simplify internal definitions (39)

  > Since `x` is bound in the definition of `sqrt`, the procedures `good-enough?`, `improve`, and `sqrt-iter`, which are defined internally to `sqrt`, are in the scope of `x`. **Thus, it is not necessary to pass `x` explicitly to each of these procedures. Instead, we allow `x` to be a free variable in the internal definitions [...]. Then `x` gets its value from the argument with which the enclosing procedure `sqrt` is called. This discipline is called *lexical scoping***.

  * A bit on how lexical scoping works (39n27)

    > Lexical scoping dictates that free variables in a procedure are taken to refer to bindings made by enclosing procedure definitions; **that is, they are looked up in the environment in which the procedure was defined**. We will see how this works in detail in chapter 3 when we study environments and the detailed behavior of the interpreter.

- A note about procedure body structure (39n28)

  > **Embedded definitions must come first in a procedure body**. The management is not responsible for the consequences of running programs that intertwine definition and use.

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

- What we've discussed so far are the *elements* of programming, but need to learn the "common patterns of usage in the domain" (40)
  - An analogy to chess (40)

    > We lack the knowledge of which moves are worth making (which procedures are worth defining). We lack the experience to predict the consequences of making a move (executing a procedure).

  - Need to gain an **ability to visualize the consequences of the actions under consideration**

    > To become experts, we **must learn to visualize the processes generated by various types of procedures**. Only after we have developed such a skill can we learn to reliably construct programs that exhibit the desired behavior.

- One sense of the word procedure (40)

A procedure is a pattern for the ***local evaluation*** of a computational process. It specifies how each stage of the process is built upon the previous stage.

  - A goal is to be able to make statements about the ***global*** behavior of process whose local evaluation has been specified by a procedure, but authors say this is a hard thing to do in general (40-1)

- In this section we'll consider some common "shapes" for processes generated by simple procedures, as well as their time and space complexity

1.2.1 Linear Recursion and Iteration (41-47)

- Highlights the difference in "shape" between a linear recursive process and a linear iterative process for factorial calculations (41-3)
  - The linear recursive process ends up building up a chain of ***deferred operations***—"a shape of expansion followed by contraction" (44)
    * Authors claim that a chain of deferred operations is characteristic of recursive processes (44)
    * "Carrying out this process **requires that the interpreter keep track of the operations to be performed later on**" (44)
    * The "linear" in "linear recursive process" refers to the fact that the number of deferred operations grows linearly with the input n
  - The linear iterative process does not grow and shrink—just have to keep track of some **state variables** (44)

    In general, an iterative process is one whose state can be summarized by a fixed number of *state variables*, together with a fixed rule that describes how the state variables should be updated as the process moves from state to state and an (optional) end test that specifies conditions under which the process should terminate.

    * Our iterative process is "linear" because the number of required grows linearly with the input n (44)

- A different contrast between iterative and recursive processes (45)

  In the iterative case, the program variables provide a complete description of the of the state of the process at any point. [...] Not so with the recursive process. In this case there is some additional "hidden" information, maintained by the interpreter and not contained in the program variables, which indicates "where the process is" in negotiation the chain of deferred operations. The longer the chain, the more information must be maintained.

- Also, need to stay clear on the fact that the notion of a recursive *process* is distinct from the notion of a recursive *procedure* (45)
  - Saying that a procedure is recursive is essentially a syntactic claim: what we're saying here is that "the procedure definition refers (either directly or indirectly) to the procedure itself" (45)
  - When we say that a process is recursive, we're speaking about how the process itself evolves, not about the syntax of how a procedure is written (45)
  - Authors note that lots of implementations of common languages (like Ada, Pascal, and C) have it that (45-6)

    > [...] the interpretation of any recursive procedure consumes an amount of memory that grows with the number of procedure calls, even when the process described is, in principle, iterative. **As a consequence, these languages can describe iterative processes only by resorting to special-purpose "looping constructs" such as `do`, `repeat`, `until`, `for`, and `while`.**

    * Authors think this is a defect of these other languages (which the implementation of Scheme discussed in Chapter 5 intentionally does not share) (46)
      - This implementation of Scheme is ***tail-recursive***, which means that it "will execute an iterative process in constant space, even if the iterative process is described by a recursive procedure" (46)
      - A note about a benefit of tail-recursive implementations (46)

        > With a tail-recursive implementation, **iteration can be expressed using the ordinary procedure call mechanism, so that special iteration constructs are useful only as syntactic sugar**.

Exercises (46-7)

- Exercise 1.9
  - (Consulted the solution presented here after getting a partial solution on my own.)

    The process generated by the first procedure, which is a *recursive* process:

    ```
    (+ 4 5)
    (inc (+ (dec 4) 5))
    (inc (+ 3 5))
    (inc (inc (+ (dec 3) 5)))
    (inc (inc (+ 2 5)))
    (inc (inc (inc (+ (dec 2) 5))))
    ```

```
(inc (inc (inc (+ 1 5))))
(inc (inc (inc (inc (+ (dec 1) 5)))))
(inc (inc (inc (inc (+ 0 5)))))
(inc (inc (inc (inc 5))))
(inc (inc (inc 6)))
(inc (inc 7))
(inc 8)
9
```

The process generated by the second procedure, which is an *iterative* process:

```
(+ 4 5)
(+ (dec 4) (inc 5))
(+ 3 6)
(+ (dec 3) (inc 6))
(+ 2 7)
(+ (dec 2) (inc 7))
(+ 1 8)
(+ (dec 1) (inc 8))
(+ 0 9)
9
```

- Exercise 1.10
  - Define the Ackermann function procedure as given in the book:

```
(define (A x y)
  (cond ((= y 0) 0)
        ((= x 0) (* 2 y))
        ((= y 1) 2)
        (else (A (- x 1) (A x (- y 1))))))
```

Then we have:

```
(A 1 10)
;Value: 1024

(A 2 4)
;Value: 65536

(A 3 3)
;Value: 65536
```

For the procedures listed in the book we have:

* `(define (f n) (A 0 n))` computes $2n$
* `(define (g n) (A 1 n))` computes 2^n
* `(define (h n) (A 2 n))` computes ${}^n2$ (a.k.a a “tetration”)

1.2.2 Tree Recursion (47-54)

- A description of **tree recursion** (48)

  In general, the evolved process looks like a tree

  - The example is given of computing the nth Fibonacci number
  - A remark on the time and space complexity of tree-recursive processes (49)

    In general, the number of steps required by a tree-recursive process will be proportional to the number of nodes in the tree, while the space required will be proportional to the maximum depth of the tree.

- Tree recursion is a particularly natural and powerful tool for processes that **operate on hierarchically structured data** rather than on numbers (50)
  - Can also be a useful cognitive tool when reasoning about particular problems—for example, if a particular problem admits both iterative and recursive processes as solutions, but the recursive process is more "natural" a solution in some sense (even if inefficient), then understanding the recursive process can sometimes be a steppingstone on the way to understanding the iterative process

Example: Counting change

Exercises (53-4)

- Exercise 1.11
  - For a procedure that computes f by means of a recursive process we can define:

    ```
    (define (f-r n)
      (cond ((< n 3) n)
            (else (+ (f (- n 1))
                     (* 2 (f (- n 2)))
                     (* 3 (f (- n 3)))))))
    ```

    And for a procedure that computes f by means of an iterative process we can define:

    ```
    (define (f-i n)
      (f-iter 2 1 0 n))

    (define (f-iter a b c count)
      (cond ((= count 0) c)
            ((= count 1) b)
    ```

```
((= count 2) a)
(else (f-iter (+ a (* 2 b) (* 3 c)) a b (- count 1)))))
```

- Exercise 1.12

```
- (define (pascal-entry row entry)
    (cond ((or (> entry row)
               (< row 1)
               (< entry 1))
           0)
          ((and (= row 1)
                (= entry 1))
           1)
          (else (+ (pascal-entry (- row 1) (- entry 1))
                   (pascal-entry (- row 1) entry)))))
```

- Exercise 1.13
  - (Might solve this and TeX the solution to put here in the future.)

1.2.3 Orders of Growth (54-57)

- Can use the notion of ***order of growth*** as a way to get a gross measure of the resources required by a process as input grows (54)
  - A seemingly standard definition of big-theta complexity is given here (55)

Exercises (56-7)

- Exercise 1.14
  - (Might take time to do this exercise in the future.)
- Exercise 1.15
  a. p is evaluated $\lceil(\log_3(\frac{12.15}{0.1}))\rceil = 5$ times.
  b. Space and number of steps both grow like $\log a$ for a given input a to the `sine` procedure.

1.2.4 Exponentiation (57-62)

- A couple implementations of an exponentiation process, and their varying orders of growth for both space and number of steps, are discussed (57-59)

Exercises (59-62)

- Exercise 1.16
  - We define an iterative `fast-expt` procedrue as follows:

```
(define (fast-expt b n)
  (fast-expt-iter b n 1))

(define (fast-expt-iter b counter product)
  (if (= counter 0)
      product
      (cond ((even? counter) (square (fast-expt-iter b (/ counter 2) product))
            (else (* b (fast-expt-iter b (- counter 1) product))))))

(define (even? n)
  (= (remainder n 2) 0))
```

- Exercise 1.17
  - We define the following `fast-mult` procedure:

```
(define (fast-mult a b)
  (cond ((= b 0) 0)
        ((even? b) (double (fast-mult a (halve b))))
        (else (+ a (fast-mult a (- b 1))))))

(define (double x) (+ x x))

(define (halve x) (/ x 2))

(define (even? n)
  (= (remainder n 2) 0))
```

- Exercise 1.18
  - Define an iterative `fast-mult`:

```
(define (fast-mult a b)
  (fast-mult-iter a b 0))

(define (fast-mult-iter a counter sum)
  (if (= counter 0)
      sum
      (cond ((even? counter) (fast-mult-iter (double a) (halve counter) sum))
            (else (fast-mult-iter a (- counter 1) (+ a sum))))))

(define (double x) (+ x x))

(define (halve x) (/ x 2))

(define (even? n)
  (= (remainder n 2) 0))
```

- Exercise 1.19

- Can see this using matrix algebra with a 2-by-2 transition matrix where the first row is $(p+q, q)$ and the second row is (q, p). (I'm not TeXing this right now because I don't know how to easily get it to show up nicely in the rendered Markdown.) If we square this matrix we have it that $p' = p^2 + q^2$ and $q' = q^2 + 2qp$.

  Now complete the procedure outlined in the book, using the values we've just derived:

```
(define (fib n)
  (fib-iter 1 0 0 1 n))

(define (fib-iter a b p q count)
  (cond ((= count 0) b)
        ((even? count)
         (fib-iter a
                   b
                   (+ (* p p) (* q q))
                   (+ (* q q) (* 2 p q))
                   (/ count 2)))
        (else (fib-iter (+ (* b q) (* a q) (* a p))
                        (+ (* b p) (* a q))
                        p
                        q
                        (- count 1)))))
```

1.2.5 Greatest Common Divisors (62-65)

- Calculating GCDs becomes important in Scheme for implementing rational-number arithmetic (62)

Exercises (65)

- Exercise 1.20
  - With normal-order evaluation we have…

```
(gcd 206 40)

(if (= 40 0) ...)

(gcd 40
     (remainder 206 40))

(if (= (remainder 206 40) 0))

(gcd (remainder 206
                40)
```

```
     (remainder 40
                (remainder 206
                           40)))

(if (= (remainder 40
                  (remainder 206
                             40))
       0) ...)

; etc.
```

...until eventually we get to an expression consisting of bunch of nested `remainder` operations, where that expression evaluates to `0`. The thing to note here is that all the `remainder` operations that are performed are those in the "final" `if` statement, and then all the `remainder` operations actually constituting the GCD calculation once the predicate of the `if` is true. Doing a quick recurrence relation, we can see that if given inputs `a` and `b` to `gcd` are comprised of n_{a_s} and n_{b_s} `remainder` operations, respectively (at step s in the recursion), then the arguments to the next recursive call of `gcd` will have $n_{a_{s+1}} = n_{b_s}$ and $n_{b_{s+1}} = n_{a_s} + n_{b_s} + 1$ `remainder` operations, respectively. (Note that this looks very similar to a Fibonacci recurrence!)

So the number of `remainder` statements generated in considering a given set of inputs looks something like, at a given step s in the process:

(a, b)	$n_{a_s} = n_{b_{s-1}}$	$n_{b_s} = n_{a_{s-1}} + n_{b_{s-1}} + 1$	total $(n_{a_s} + n_{b_s})$
(206, 40)	0	1	1
(40, 6)	1	2	3
(6, 4)	2	4	6
(4, 2)	4	7	11

After this last phase, we reach the base case of the recursion—at which point 7 `remainder` operations are performed in the predicate of the `if` statement, and 4 are performed in the consequent expression (and none in the alternative expression, by the rule for normal-order `if` cited in Exercise 1.5). So, in total, 11 `remainder` operations are performed in normal-order evaluation.

In applicative-order evaluation, the process looks different:

```
(gcd 206 40)
(if (= 40 0) ...)

(gcd 40 (remainder 206 40))
(gcd 40 6)
(if (= 6 0) ...)
```

```
(gcd 6 (remainder 40 6))
(gcd 6 4)
(if (= 4 0) ...)

(gcd 4 (remainder 6 4))
(gcd 4 2)
(if (= 2 0) ...)

(gcd 2 (remainder 4 2))
(gcd 2 0)
(if (= 0 0) ...)

2
```

So applicative-order evaluation performs 4 `remainder` operations in the evaluation of `(gcd 206 40)`.

1.2.6 Example: Testing for Primality (65-74)

- Sumnmary of the section (65)

  This section describes two methods for checking the primality of an integer n, one with order of growth $\Theta(\sqrt{n})$ and a "probabilistic" algorithm with order of growth $\Theta(\log n)$. The exercises at the end of this section suggest programming projects based on these algorithms.

Searching for divisors (65-6)

- One way to test if a number is prime is to find that number's divisors (65)
  - And so can implement a procedure (called `prime?` in the book) that tests if a given n is it's own smallest divisor (which is equivalent condition to n being prime) (66)
  - The implementation in the book only tests numbers up through $\sqrt{n}$ for a given input n—hence a $\Theta(\sqrt{n})$ order of growth

The Fermat test (66-8)

- A $\Theta(\log n)$ result primality test can be derived using Fermat's Little Theorem (66-7)

  Fermat's Little Theorem: If n is a prime number and a is any positive integer less than n, then a raised to the n^{th} power is congruent to a modulo n.

- The relevant algorithm here is the **Fermat test** (67)

> If n is not prime, then, in general, most of the numbers $a < n$ will not satisfy [the relation described in Fermat's Little Theorem]. This leads to the following algorithm for testing primality: Given a number n, pick a random number $a < n$ and compute the remainder of a^n modulo n. If the result is not equal to a, then n is certainly not prime. If it is a, then chances are good that n is prime. Now pick another random number a and test it with the same method. If it also satisfies the equation, then we can be even more confident that n is prime. By trying more and more values of a, we can increase our confidence in the result. **This algorithm is known as the Fermat test**.

  - This algorithm involves the **expmod** operation
    * Implementation details are given in the book; the crucial implementation detail for the logarithmic order of growth has to do with the implementation of `expmod` (given on page 67)

Probabilistic methods (69-70)

- Gives some discussion of probabilistic (non-)guarantees of correctness, as well as alternative tests that can get around numbers that can "fool" the ordinary Fermat test (i.e. *Carmichael numbers*) (69, 69n47)

Exercises (70-4)

- Exercise 1.21
  - Define the `smallest-divisor` procedure (and its dependencies) as given in the text on pp. 65-6:

```
(define (smallest-divisor n) (find-divisor n 2))

(define (find-divisor n test-divisor)
  (cond ((> (square test-divisor) n) n)
        ((divides? test-divisor n) test-divisor)
        (else (find-divisor n (+ test-divisor 1)))))

(define (divides? a b) (= (remainder b a) 0))
```

  Now we use `smallest-divisor` to find the smallest divisor of each of 199, 1999, and 19999:

```
(smallest-divisor 199)

;Value: 199


(smallest-divisor 1999)
```

```
;Value: 1999


(smallest-divisor 19999)

;Value: 7
```

- Exercise 1.22
  - First define the procedures given in the text on pg. 70 (slightly modified to only display n if n is prime), along with the procedure `prime?` from pp. 65-6:

```
(define (timed-prime-test n)
  (start-prime-test n (runtime)))

(define (start-prime-test n start-time)
  (if (prime? n)
    (report-prime (- (runtime) start-time) n)))

(define (report-prime elapsed-time n)
  (newline)
  (display n)
  (display " *** ")
  (display elapsed-time))
```

```
(define (prime? n)
  (= n (smallest-divisor n)))

(define (smallest-divisor n) (find-divisor n 2))

(define (find-divisor n test-divisor)
  (cond ((> (square test-divisor) n) n)
        ((divides? test-divisor n) test-divisor)
        (else (find-divisor n (+ test-divisor 1)))))

(define (divides? a b) (= (remainder b a) 0))
```

And now define our `search-for-primes` procedure (consulted with the solution at the sicp-wiki for some implementation details):

```
(define (search-for-primes low high)
  (define (search-iter cur last)
    (if (<= cur last)
        (timed-prime-test cur))
    (if (and (<= cur last))
        (search-iter (+ cur 2) last)))
  (search-iter (if (= (remainder low 2) 0) (+ low 1) low)
               (if (= (remainder high 2) 0) (- high 1) high)))
```

This is the procedure called for by the book—but to answer the question posed, a slightly different procedure would be more helpful. Call this procedure `get-first-n-primes-greater`:

```
(define (get-first-n-primes-greater low n)
  (define (search-iter cur count max-count)
    (timed-prime-test cur)
    (if (and (< count n)) ; leverage lexical scoping here
        (search-iter (+ cur 2 )
                     (if (prime? cur)
                         (+ count 1)
                         count)
                     max-count)))
  (search-iter (if (= (remainder low 2) 0) (+ low 1) low)
               0
               n))
```

Now let's try out the procedure for sets of arguments that differ in order of magnitude

```
(get-first-n-primes-greater 1000000000 3)    ;1e9

1000000007 *** 3.0000000000001137e-2
1000000009 *** 1.9999999999999574e-2
1000000021 *** 3.0000000000001137e-2
;Unspecified return value

(get-first-n-primes-greater 10000000000 3)   ;1e10

10000000019 *** .0799999999999983
10000000033 *** .08000000000000185
10000000061 *** .07000000000000028
;Unspecified return value

(get-first-n-primes-greater 100000000000 3)  ;1e11

100000000003 *** .22999999999999687
100000000019 *** .23999999999999844
100000000057 *** .23000000000000043
;Unspecified return value

(get-first-n-primes-greater 1000000000000 3) ;1e12

1000000000039 *** .7400000000000002
1000000000061 *** .7199999999999989
1000000000063 *** .7199999999999989
;Unspecified return value
```

Dividing some candidate values from each successive magnitude of input

gives `elapsed-time` ratios of between 2.7 and 3.2—which aligns (more or less) with the expected with the $\Theta(\sqrt{10})$ order of growth.

- Exercise 1.23
  - Define a new `find-divisor` procedure as follows:

```
(define (find-divisor n test-divisor)
  (cond ((> (square test-divisor) n) n)
        ((divides? test-divisor n) test-divisor)
        (else (find-divisor n (next test-divisor)))))

(define (next n)
  (cond ((= n 2) 3)
        (else (+ n 2))))
```

Now we have

```
(get-first-n-primes-greater 1000000000 3)    ;1e9

1000000007 *** 2.0000000000000018e-2
1000000009 *** 9.999999999999787e-3
1000000021 *** 9.999999999999787e-3
;Unspecified return value

(get-first-n-primes-greater 10000000000 3)   ;1e10

10000000019 *** 4.0000000000000036e-2
10000000033 *** .04999999999999982
10000000061 *** 4.0000000000000036e-2
;Unspecified return value

(get-first-n-primes-greater 100000000000 3)  ;1e11

100000000003 *** .14000000000000057
100000000019 *** .15000000000000036
100000000057 *** .13999999999999968
;Unspecified return value

(get-first-n-primes-greater 1000000000000 3) ;1e12

1000000000039 *** .4500000000000002
1000000000061 *** .45999999999999996
1000000000063 *** .4500000000000002
;Unspecified return value
```

So the execution times for corresponding primes are (roughly) anywhere between 1.6x and 3x faster for the alternative implementation of `find-divisor`. This is certainly faster, but it seems like there might be work

going on under the hood in the computer that impacts the execution time of the programs, beyond just how we define our procedure.

- Exercise 1.24
  - After doing some tests on many orders of magnitude of inputs to `get-first-n-primes-greater` implemented using `fast-prime?` it seems like the order of growth is… kind of logarithmic? E.g.,

```
(get-first-n-primes-greater #e1e100 3)

100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
;Unspecified return value

(get-first-n-primes-greater #e1e200 3)

100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
;Unspecified return value
```

  At least, judging by the last prime found by each procedure call, where a prime close to 1e200 digits takes about twice as long as a prime close to 1e100.

  (Maybe I don't have the implementation quite right here. This implementation is at exercise_1_24.txt.)

- Exercise 1.25
  - This alternative implementation would not serve well because it would suffer in performance for very large numbers (because `fast-expt` would have sometimes have to calculate massive exponentials). See pg. 68 footnote 46 for an explanation of why our original `expmod` implementation is able to handle finding very large primes.

- Exercise 1.26
  - This is due to the behavior of applicative-order evaluation: under the implementation of explicit multiplication, the number of times `(expmod base (/ exp 2) m)` is called grows like $\Theta(2^{\log n})$, which is equivalent to $\Theta(n)$. (That is, there's a sort of branching binary tree structure to the recursive calls (i.e., a tree recursion) of `expmod` under this interpretation—and so even though that tree is only $\log n$ deep, each "level" of the tree has a number of nodes that is greater than the level above it by a factor of 2.) On the other hand, using a `square` procedure means that each recursive `expmod` procedure is only evaluated once at a given level in the recursion (and so it's a linear recursion), which keeps things to a $\Theta(\log n)$ order of growth (since the recursion is only $\log n$ steps deep).

- Exercise 1.27
  - Define a procedure `fermat-test-all`:

```
(define (fermat-test-all n)
  (define (fermat-test-all-iter a)
    (cond ((>= a n) (display "passed"))
          ((try-it a) (fermat-test-all-iter (+ a 1)))
          (else (display "fermat test failed"))))
  (define (try-it a)
    (= (expmod a n n) a))
  (fermat-test-all-iter 1))
```

For a given n, this procedure will display `passed` if a^n is congruent to a modulo n for every $a < n$, and will display `fermat test failed` else. We can check our `fermat-test-all` procedure on the given the Carmichael numbers, and also verify that these numbers aren't primes using the `prime?` procedure defined in the text:

```
(fermat-test-all 561)
passed
;Unspecified return value

(prime? 561)
;Value: #f

(fermat-test-all 1105)
passed
;Unspecified return value

(prime? 1105)
;Value: #f

(fermat-test-all 1729)
passed
;Unspecified return value

(prime? 1729)
;Value: #f

(fermat-test-all 2465)
passed
;Unspecified return value

(prime? 2465)
;Value: #f

(fermat-test-all 2821)
passed
;Unspecified return value
```

```
(prime? 2821)
;Value: #f

(fermat-test-all 6601)
passed
;Unspecified return value

(prime? 6601)
;Value: #f
```

- Exercise 1.28
  - Define an expmod procedure that makes used of an alternative squaring procedure that will detect nontrivial square roots of 1, called `square-detect-nontriv`:

```
(define (expmod-detect-nontriv base exp m)
  (define (square-detect-nontriv x)
    (define (square x) (* x x))
    (if (and (not (= x 1))
             (not (= x (- m 1)))
             (= (remainder (square x) m) 1))
        0 ;signal value
        (square x)))
  (cond ((= exp 0) 1)
        ((even? exp)
         (remainder
          (square-detect-nontriv (expmod-detect-nontriv base (/ exp 2) m))
          m))
        (else
         (remainder
          (* base (expmod-detect-nontriv base (- exp 1) m))
          m))))
```

And now define a `miller-rabin-test` procedure that makes use of the `expmod-detect-nontriv` procedure:

```
(define (miller-rabin-test n)
  (define (miller-rabin-test-iter a)
    (cond ((>= a n) (display "passed"))
          ((try-it a) (miller-rabin-test-iter (+ a 1)))
          (else (display "miller-rabin test failed"))))
  (define (try-it a)
    (= (expmod-detect-nontriv a n n) a))
  (miller-rabin-test-iter 1))
```

Let's try out `miller-rabin-test` on some primes…

```
(miller-rabin-test 2)
```

```
passed
;Unspecified return value

(miller-rabin-test 7)
passed
;Unspecified return value

(miller-rabin-test 13)
passed
;Unspecified return value
```

...some non-primes...

```
(miller-rabin-test 4)
miller-rabin test failed
;Unspecified return value

(miller-rabin-test 63)
miller-rabin test failed
;Unspecified return value

(miller-rabin-test 100)
miller-rabin test failed
;Unspecified return value
```

...and some tricky non-primes:

```
(miller-rabin-test 561)
miller-rabin test failed
;Unspecified return value

(miller-rabin-test 1105)
miller-rabin test failed
;Unspecified return value

(miller-rabin-test 1729)
miller-rabin test failed
;Unspecified return value

(miller-rabin-test 2465)
miller-rabin test failed
;Unspecified return value

(miller-rabin-test 2821)
miller-rabin test failed
;Unspecified return value

(miller-rabin-test 6601)
miller-rabin test failed
;Unspecified return value
```

So it seems like our `miller-rabin-test` procedure correctly implements the Miller-Rabin test.

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

- A framing thought for what we've seen so far (74)

  > We have seen that **procedures are, in effect, abstractions that describe compound operations on numbers independent of the particular numbers**.

  - A framing of procedures as "allowing our language to express the concept of [something]" (75)

- A desideratum of "powerful" programming languages (75)

  > One of the things we should demand from a powerful programming language is the **ability to build abstractions by assigning names to common patterns and then to work in terms of the abstractions directly**. Procedures provide this ability. This is why all but the most primitive programming languages include mechanisms for defining procedures.

- ***Higher-order procedures*** and their motivatiopn (75)

  > Often the same programming pattern will be used with a number of different procedures. To express such patterns as concepts we will need to construct procedures that can accept procedures as arguments or return procedures as values. **Procedures that manipulate procedures are called *higher-order procedures***.

1.3.1 Procedures as Arguments (76-82)

- Some examples are given of summation of series (77)
  - In math, "sigma notation" (77)

    > The power of sigma notation is that it allows mathematicians to deal with the concept of summation itself rather than only with particular sums.

- In practice, we can just pass in the names of procedures as arguments to other procedures (77-8)

Exercises (80-2)

- Exercise 1.29
  - First, define the `sum` procedure as given on pp. 77-8 in the text:

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b))))
```

Now define a procedure `integrate-simpson` that implements numerical integration using Simpson's Rule:

```
(define (integrate-simpson f a b n)
  (define (simpson-next k)
    (+ k 1))
  (define (simpson-term k)
    (cond ((or (= k 0) (= k n)) (f (+ a (* k (/ (- b a) n)))))
          ((odd? k) (* 4 (f (+ a (* k (/ (- b a) n))))))
          (else (* 2 (f (+ a (* k (/ (- b a) n))))))))
  (* (/ (- b a) n 3) (sum simpson-term 0 simpson-next n)))
```

Now let's see how the values turn out when integrating `cube` between 0 and 1, for $n = 100$ and $n = 1000$:

```
(integrate-simpson cube 0 1 100)

;Value: 1/4

(integrate-simpson cube 0 1 1000)

;Value: 1/4
```

So `integrate-simpson` gives (and *can* give) answers that are exactly correct. Compare this to the results of the `integral` procedure defined on pg. 79 of the text, where we only ever get approximately correct (even if very, very accurate) solutions. This is likely due to how Scheme represents floating point numbers vs. how it represents rational numbers (and how the procedures we've defined make use of these various representations of numbers).

- Exercise 1.30
  - We can define an iterative `sum` procedure as follows:

```
(define (sum term a next b)
  (define (iter a result)
    (if (> a b)
        result
        (iter (next a) (+ result (term a)))))
  (iter a 0))
```

- Exercise 1.31

  a. Define our `product` procedure as follows:

```
(define (product term a next b)
  (if (> a b)
      1
      (* (term a)
         (product term (next a) next b))))
```

`factorial` can be implemented as follows:

```
(define (factorial n)
  (define (identity x) x)
  (define (inc x) (+ x 1))
  (product identity 1 inc n))
```

We can implement a procedure to approximate π—call it `pi-approx`—as follows:

```
(define (pi-approx n)
  (define (next-largest-even x)
    (if (even? x) (+ x 2) (+ x 1)))
  (define (next-largest-odd x)
    (if (odd? x) (+ x 2) (+ x 1)))
  (define (frac-term m)
    (/ (next-largest-even m) (next-largest-odd m)))
  (define (inc x) (+ x 1))
  (* 4.0 (product frac-term 1 inc n)))
```

Looking at the value of `pi-approx` with an input value of $n = 10000$ can help us validate our implementation:

```
(pi-approx 10000)

;Value: 3.1417497057380523
```

b. Implement an iterative `product` procedure as follows:

```
(define (product term a next b)
  (define (iter a result)
    (if (> a b)
        result
        (iter (next a) (* result (term a)))))
  (iter a 1))
```

Testing the iterative implementation of `product` with a couple procedures and input values whose corresponding output values we know already:

```
(factorial 5)

;Value: 120

(pi-approx 10000)

;Value: 3.1417497057380523
```

And so we see we get back exactly the same values as we did with the recursive implementation of `product`.

- Exercise 1.32

  a. Define a procedure `accumulate` as follows:

```
(define (accumulate combiner null-value term a next b)
  (if (> a b)
      null-value
      (combiner (term a)
         (accumulate combiner null-value term (next a) next b))))
```

Now define `sum` in terms of `accumulate`:

```
(define (sum term a next b)
  (accumulate + 0 term a next b))
```

And `product`:

```
(define (product term a next b)
  (accumulate * 1 term a next b))
```

b. Define an iterative `accumulate` procedure:

```
(define (accumulate combiner null-value term a next b)
  (define (iter a result)
    (if (> a b)
        result
        (iter (next a) (combiner result (term a)))))
  (iter a null-value))
```

- Exercise 1.33

  a. Define `filtered-accumulate` as follows:

```
(define (filtered-accumulate predicate combiner null-value term a next b)
  (if (> a b)
      null-value
      (combiner (if (predicate a) (term a) null-value)
         (filtered-accumulate predicate combiner null-value term (next a) next b))))
```

  Now define `ssq-primes` as follows (making use of the `prime?` procedure from pp. 65-6 of the text):

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

(define (inc x) (+ x 1))

(define (ssq-primes a b)
  (filtered-accumulate prime? + 0 square a inc b))
```

  b. Define `product-lt-rel-prime` as follows (making use of the `gcd` procedure defined on pg. 63 of the text):

```
(define (product-lt-rel-prime n)
  (define (rel-prime? i)
    (= (gcd i n) 1))
  (define (identity x) x)
```

```
(define (inc x) (+ x 1))
(filtered-accumulate rel-prime? * 1 identity 1 inc (- n 1)))
```

1.3.2 Constructing Procedures Using `lambda` (83-88)

- As opposed to naming procedures, can use the **special form `lambda`** to specify "the procedure that returns…" (83)
  - General structure of the lambda special form: (84)

    ```
    (lambda ([formal parameters]) [body])
    ```

    e.g. (from pg. 84 of the text),

    ```
    (lambda (x y z) (+ x y (square z)))
    ```

- Relationship of `lambda` to `define` (84)

  > The resulting procedure [created by the *lambda* special form] is just as much a procedure as one that is created using *define*. **The only difference is that it has not been associated with any name in the environment**. In fact,
  >
  > ```
  > (define (plus4 x) (+ x 4))
  > ```
  >
  > is equivalent to
  >
  > ```
  > (define plus4 (lambda (x) (+ x 4)))
  > ```

- Can read `lambda` expressions like (for the example of `(lambda (x) (+ x 4))`): "The procedure (`lambda`) of an argument x (x) that adds (+) x (x) and 4 (4)" (84)
- Since `lambda` expressions have procedures as their values, they can be used as the operators of combinations (or, more generally, in any context where we would normally use a procedure name) (84)

Using `let` to create local variables (85-8)

- `lambda` can also be used to create **local variables**—and **so can the special form `let`** (85)
  - The general form of `let`: (86)

    ```
    (let (([var_1] [exp_1])
          ([var_2] [exp_2])
          ...
          ([var_n] [exp_n]))
      [body])
    ```

    Under the hood, this is interpreted as alternate syntax for an expression involving `lambda`: (86-7)

```
((lambda ([var_1] ... [var_n])
  [body])
[exp_1]
...
[exp_n])
```

- A point made in the book about how `let` is alternate syntax for a certain kind of expression involving `lambda`: (87)

  > **No new mechanism is required in the interpreter in order to provide local variables. A `let` expression is simply syntactic sugar for the underlying `lambda` application.**

- From the equivalence we can see that **the scope of a variable specified in a `let` expression is the body of the `let`** (87)
  - What this implies (87-8)
    - `let` allows one to bind variables as locally as possible to where they are to be used
    - The variables' values are computed outside the `let`, which matters when the expressions that provide the values for the local variables depend on variables with the same names as the local variables themselves
- Could in principle use internal definitions to get the same effect as a `let`—but as a matter of style, the authors prefer to use internal `define` statements for internal procedures only (88)

Exercises (88)

- Exercise 1.34
  - The process evolves as follows:

    ```
    (f f)

    (f 2)

    (2 2)
    ```

    Which throws an error:

    ```
    ;The object 2 is not applicable.
    ```

    This error arises because in `(2 2)`, the interpreter tries to treat `2` as the name of an operator in a combination, which can't happen given how `2` is defined in the environment.

1.3.3 Procedures as General Methods (89-96)

Finding roots of equations by the half-interval method (89-91)

Finding fixed points of functions (92-4)

- (There's a note about *average damping* on pg. 94. This is my first time learning about this idea!)

Exercises (94-6)

- Exercise 1.35
  - Define the `fixed-point` procedure given in the text on pg. 92:

```
(define tolerance 0.00001)
(define (fixed-point f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2))
       tolerance))
   (define (try guess)
    (let ((next (f guess)))
      (if (close-enough? guess next)
        next
        (try next))))
  (try first-guess))
```

    Now compute the golden ratio ϕ using a fixed point transformation of $f(x) = 1 + \frac{1}{x}$:

```
(fixed-point (lambda (x) (+ 1 (/ 1 x))) 2.0)

;Value: 1.6180327868852458
```

- Exercise 1.36
  - Define the modified `fixed-point` procedure as follows:

```
(define tolerance 0.00001)
(define (fixed-point f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2))
       tolerance))
   (define (try guess)
    (newline)
    (display guess)
    (let ((next (f guess)))
      (if (close-enough? guess next)
        next
        (try next))))
  (try first-guess))
```

Now run the procedure:

```
(fixed-point (lambda (x) (/ (log 1000) (log x))) 2.0)

2.
9.965784284662087
3.004472209841214
6.279195757507157
3.759850702401539
5.215843784925895
4.182207192401397
4.8277650983445906
4.387593384662677
4.671250085763899
4.481403616895052
4.6053657460929
4.5230849678718865
4.577114682047341
4.541382480151454
4.564903245230833
4.549372679303342
4.559606491913287
4.552853875788271
4.557305529748263
4.554369064436181
4.556305311532999
4.555028263573554
4.555870396702851
4.555315001192079
4.5556812635433275
4.555439715736846
4.555599009998291
4.555493957531389
4.555563237292884
4.555517548417651
4.555547679306398
4.555527808516254
4.555540912917957
;Value: 4.555532270803653
```

So this took 34 steps (if I counted correctly!). Let's see how many steps things take with average damping:

```
(fixed-point (lambda (x) (/ (+ x (/ (log 1000) (log x))) 2)) 2.0)

2.
5.9828921423310435
```

```
4.922168721308343
4.628224318195455
4.568346513136242
4.5577305909237005
4.555909809045131
4.555599411610624
4.5555465521473675
;Value: 4.555537551999825
```

So with average damping, our fixed point process converges in 9 steps.

- Exercise 1.37

a. Define `count-frac` as follows:

```
(define (cont-frac n d k)
  (define (recurse i)
    (if (> i k)
        0
        (/ (n i) (+ (d i) (recurse (+ i 1))))))
  (recurse 1))
```

Now let's check the procedure by approximating $\frac{1}{\phi}$ using the procedure given in the text on pg. 95 (using `10` as our value for `k`):

```
(cont-frac (lambda (i) 1.0)
           (lambda (i) 1.0)
           10)

;Value: .6179775280898876
```

And we see that the reciprocal of the value obtained is approximately equal to the golden ratio:

```
(/ 1 .6179775280898876)

;Value: 1.6181818181818184
```

We can get an approximation that's accurate to four decimal places by setting `k` equal to `12`:

```
(/ 1
   (cont-frac (lambda (i) 1.0)
              (lambda (i) 1.0)
              12))
```

b. Define an iterative `cont-frac` procedure as follows:

```
(define (cont-frac n d k)
  (define (iter i result)
    (if (< i 1)
        result
```

```
      (iter (- i 1) (/ (n i) (+ (d i) result)))))
  (iter k 0))
```

And let's check this implementation against a known answer from part (a):

```
(cont-frac (lambda (i) 1.0)
           (lambda (i) 1.0)
           10)

;Value: .6179775280898876
```

- Exercise 1.38
  - Define a procedure `approx-e` as follows:

    ```
    (define (approx-e k)
      (define (n i) 1.0)
      (define (d i)
        (if (= (remainder i 3) 2)
            (* 2.0 (/ (+ i 1) 3))
            1.0))
      (+ 2 (cont-frac n d k)))
    ```

    And test it out:

    ```
    (approx-e 100)

    ;Value: 2.7182818284590455
    ```

- Exercise 1.39
  - Define the `tan-cf` procedure as follows:

    ```
    (define (tan-cf x k)
      (cont-frac (lambda (i) (if (= i 1) x (* -1 x x)))
                 (lambda (i) (- (* i 2) 1.0))
                            k))
    ```

1.3.4 Procedures as Returned Values (97-106)

- A note about designing procedures (98)

  > In general, there are many ways to formulate a process as a procedure. Experienced programmers know how to choose procedural formulations that are particularly perspicuous, and where useful elements of the process are exposed as separate entities that can be reused in other applications.

Newton's method (98-100)

Abstractions and first-class procedures (101-3)

- Just adding a couple good paragraphs here verbatim (101-2)

  > We began Section 1.3 with the observation that compound procedures are a crucial abstraction mechanism, because they permit us to express general methods of computing as explicit elements in our programming language. Now we've seen how higher-order procedures permit us to manipulate these general methods to create further abstractions.
  >
  > As programmers, we should be alert to opportunities to identify the underlying abstractions in our programs and to build upon them and generalize them to create more powerful abstractions. This is not to say that one should always write programs in the most abstract way possible; expert programmers know how to choose the level of abstraction appropriate to their task. But it is important to be able to think in terms of these abstractions, so that we can be ready to apply them in new contexts. The significance of higher-order procedures is that they enable us to represent these abstractions explicitly as elements in our programming language, so that they can be handled just like other computational elements.
  >
  > In general, programming languages impose restrictions on the ways in which computational elements can be manipulated. Elements with the fewest restrictions are said to have *first-class* status. Some of the "rights and privileges" of first-class elements are:
  >
  > – They may be named variables.
  > – They may be passed as arguments to procedures.
  > – They may be returned as the results of procedures.
  > – They may be included in data structures.
  >
  > Lisp, unlike other common programming languages, awards procedures full first-class status. This poses challenges for efficient implementation, but the resulting gain in expressive power is enormous.

Exercises (103-6)

- Exercises 1.41
  - [TODO]
- Exercises 1.42
  - [TODO]
- Exercises 1.43
  - [TODO]
- Exercises 1.44

    - [TODO]
- Exercises 1.45
    - [TODO]
- Exercises 1.46
    - [TODO]