A Journey Through SICP

Notes, exercises and analyses of Abelson and Sussman

ProducerMatt

September 19, 2022

Contents

1 Introduction Notes

1.1 Text Foreword

This book centers on three areas: the human mind, collections of computer programs, and the computer.

Every program is a model of a real or mental process, and these processes are at any time only partially understood. We change these programs as our understandings of these processes evolve.

Ensuring the correctness of programs becomes a Herculean task as complexity grows. Because of this, it's important to make fundamentals that can be relied upon to support larger structures.

1.2 Preface, 1e

"Computer Science" isn't really about computers or science, in the same way that geometry isn't really about measuring the earth ('geometry' translates to 'measurement of earth').

Programming is a medium for expressing ideas about methodology. For this reason, programs should be written first for people to read, and second for machines to execute.

The essential material for introductory programming is how to control complexity when building programs.

Computer Science is about imperative knowledge, as opposed to declarative. This can be called *procedural epistemology*.

Declarative knowledge *what is true*. For example: $\sqrt{x}$ is the y such that $y^2 = x$ and $y \geq 0$

Imperative knowledge *How to follow a process*. For example: to find an approximation to $\sqrt{x}$, make a guess G, improve the guess by averaging G and x/G, keep improving until the guess is good enough.

1. Techniques for controlling complexity

   Black-box abstraction Encapsulating an operation so the details of it are irrelevant.

   The fixed point of a function $f()$ is a value y such that $f(y) = y$. Method for finding a fixed point: start with a guess for y and keep applying $f(y)$ over and over until the result doesn't change very much.

   Define a box of the method for finding the fixed point of $f()$.

   One way to find $\sqrt{x}$ is to take our function for approaching a square root `(λ(guess target) (average guess (divide target guess)))`, applying that to our method for finding a fixed point, and this creates a **procedure** to find a square root.

   Black-box abstraction

   (a) Start with primitive objects of procedures and data.
   (b) Combination: combine procedures with *composition*, combine data with *construction* of compound data.
   (c) Abstraction: defining procedures and abstracting data. Capture common patterns by making high-order procedures composed of other procedures. Use data as procedures.

   Conventional interfaces Agreed-upon ways of connecting things together.

   - How do you make operations generalized?
   - How do you make large-scale structure and modularity?

     Object-oriented programming thinking of your structure as a society of discrete but interacting parts.

     Operations on aggregates thinking of your structure as operating on a stream, comparable to signal processing. *(Needs clarification.)*

   Metalinguistic abstractions Making new languages. This changes the way you interact with the system by letting you emphasize some parts and deemphasize other parts.

2 Chapter 1: Building Abstractions with Procedures

Computational processes are abstract 'beings' that inhabit computers. Their evolution is directed by a pattern of rules called a **program**, and processes manipulate other abstract things called **data**.

Master software engineers are able to organize programs so they can be reasonably sure the resulting process performs the task intended, without catastrophic consequences, and that any problems can be debugged.

Lisp's users have traditionally resisted attempts to select an "official" version of the language, which has enabled Lisp to continually evolve.

There are powerful program-design techniques which rely on the ability to blur the distinction between data and processes. Lisp enables these techniques by allowing processes to be represented and manipulated as data.

2.1 1.1: The Elements of Programming

A programming language isn't just a way to instruct a computer – it's also a framework for the programmer to organize their ideas. Thus it's important to consider the means the language provides for combining ideas. Every powerful language has three mechanisms for this:

primitive expressions the simplest entities the language is concerned with

means of combination how compound elements can be built from simpler ones

means of abstraction how which compound elements can be named and manipulated as units

In programming, we deal with **data** which is what we want to manipulate, and **procedures** which are descriptions of the rules for manipulating the data.

A procedure has **formal parameters**. When the procedure is applied, the formal parameters are replaced by the **arguments** it is being applied to. For example, take the following code:

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

```
<<square>>
(square 5)
```

`x` is the formal parameter and `5` is the argument.

2.2 1.1.1: Expressions

The general form of Lisp is evaluating **combinations**, denoted by parenthesis, in the form `(operator operands)`, where *operator* is a procedure and *operands* are the 0 or more arguments to the operator.

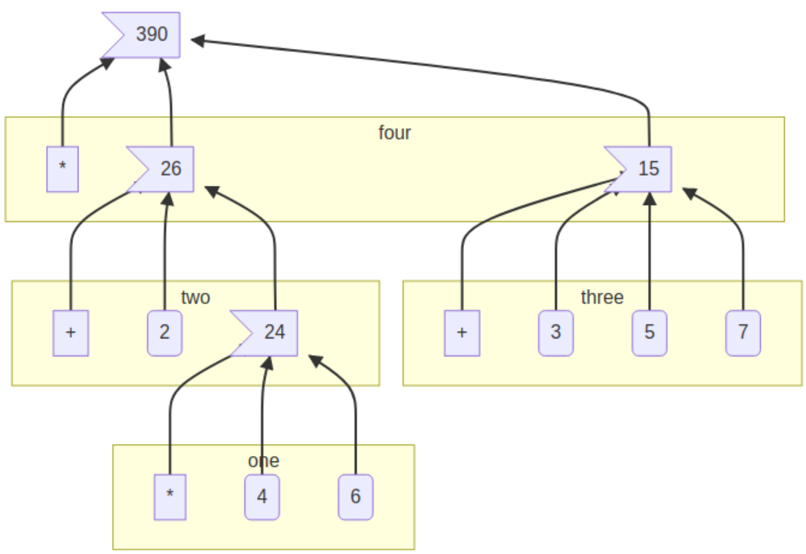
Lisp uses **prefix notation**, which is not customary mathematical notation, but provides several advantages.

1. It supports procedures that take arbitrary numbers of arguments, i.e. `(+ 1 2 3 4 5)`.
2. It's straightforward to nest combinations in other combinations.

2.3 1.1.3: Evaluating Combinations

The evaluator can evaluate nested expressions recursively. **Tree accumulation** is the process of evaluating nested combinations, "percolating" values upward.

The recursive evaluation of `(* (+ 2 (* 4 6)) (+ 3 5 7))` breaks down into four parts:



2.4 1.1.4: Compound Procedures

We have identified the following in Lisp:

- primitive data are numbers, primitive procedures are arithmetic operations
- Operations can be combined by nesting combinations
- Data and procedures can be abstracted by variable & procedure definitions

Procedure definitions give a name to a compound procedure.

```
(define (square x) (* x x)) ; to square something, multiply it by itself
; now it can be applied or used in other definitions:
(square 4) ; => 16

(define (sum-of-squares x y)
  (+ (square x) (square y)))
(sum-of-squares 3 4) ; => 25
```

Note how these compound procedures are used in the same way as primitive procedures.

2.5 1.1.5: The Substitution Model for Procedure Application

To understand how the interpreter works, imagine it substituting the procedure calls with the bodies of the procedure and its arguments.

```
(* (square 3) (square 4))
; has the same results as
(* (* 3 3) (* 3 3))
```

This way of understanding procedure application is called the **substitution model**. This model is to help you understand procedure substitution, and is usually not how the interpreter actually works. This book will progress through more intricate models of interpreters as it goes. This is the natural progression when learning scientific phenomena, starting with a simple model, and replace it with more refined models as the phenomena is examined in more detail.

Evaluations can be done in different orders.

Applicative order evaluates the operator and operands, and then applies the resulting procedure to the resulting arguments. In other words, reducing, then expanding, then reducing.

Normal order substitutes expressions until it obtains an expression involving only primitive operators, or until it can't substitute any further, and then evaluates. This results in expanding the expression completely before doing any reduction, which results in some repeated evaluations.

For all procedure applications that can be modeled using substitution, applicative and normal order evaluation produce the same result. Normal order becomes more complicated once dealing with procedures that can't be modeled by substitution.

Lisp uses applicative order evaluation because it helps avoid repeated work and other complications. But normal has its own advantages which will be explored in Chapter 3 and 4.

```
; Applicative evaluation
(f 5)
(sum-of-squares (+ a 1) (* a 2))
(sum-of-squares (+ 5 1) (* 5 2))
(sum-of-squares 6 10)
(+ (square x)(square y))
(+ (square 6)(square 10))
(+ (* 6 6)(* 10 10))
(+ 36 100)
136
; Normal evaluation
(f 5)
(sum-of-squares (+ a 1) (* a 2))
(sum-of-squares (+ 5 1) (* 5 2))
(+ (square (+ 5 1)) (square (* 5 2)))
(+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2)))
(+ (* 6 6) (* 10 10))
(+ 36 100)
136
```

(Extra-curricular clarification: Normal order delays evaluating arguments until they're needed by a procedure, which is called lazy evaluation.)

2.6 1.1.6: Conditional Expressions and Predicates

An important aspect of programming is testing and branching depending on the results of the test. `cond` tests **predicates**, and upon encountering one, returns a **consequent**.

```
(cond
     (predicate1 consequent1)
     ...
     (predicateN consequentN))
```

A shorter form of conditional:

```
(if predicate consequent alternative)
```

If `predicate` is true, `consequent` is returned. Else, `alternative` is returned.
Combining predicates:

```
(and expression1 ... expressionN)
; if encounters false, stop eval and returns false.
(or expression1 ... expressionN)
; if encounters true, stop eval and return true. Else false.
(not expression)
```

```
; true is expression is false, false if expression is true.
```

A small clarification:

```
(define A (* 5 5))
(define (D) (* 5 5))
A ; => 25
D ; => compound procedure D
(D) ; => 25 (result of executing procedure D)
```

Special forms bring more nuances into the substitution model mentioned previously. For example, when evaluating an `if` expression, you evaluate the predicate and, depending on the result, either evaluate the **consequent** or the **alternative**. If you were evaluating in a standard manner, the consequent and alternative would both be evaluated, rendering the `if` expression ineffective.

2.7 Exercise 1.1

2.7.1 Question

> Below is a sequence of expressions. What is the result printed by the interpreter in response to each expression? Assume that the sequence is to be evaluated in the order in which it is presented.

2.7.2 Answer

```
10 ;; 10
(+ 5 3 4) ;; 12
(- 9 1) ;; 8
(/ 6 2) ;; 3
(+ (* 2 4) (- 4 6)) ;; 6
(define a 3) ;; a=3
(define b (+ a 1)) ;; b=4
(+ a b (* a b)) ;; 19
(= a b) ;; false
(if (and (> b a) (< b (* a b)))
    b
    a) ;; 4
(cond ((= a 4) 6)
      ((= b 4) (+ 6 7 a))
      (else 25)) ;; 16
(+ 2 (if (> b a) b a)) ;; 6
(* (cond ((> a b) a)
         ((< a b) b)
         (else -1))
   (+ a 1)) ;; 16
```

2.8 Exercise 1.2

2.8.1 Question

Translate the following expression into prefix form:

$$\frac{5 + 2 + (2 - 3 - (6 + \frac{4}{5}))}{3(6 - 2)(2 - 7)}$$

2.8.2 Answer

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

```
1/75
```

2.9 Exercise 1.3

2.9.1 Question

Define a procedure that takes three numbers as arguments and returns the sum of the squares of the two larger numbers.

2.9.2 Answer

```
<<square>>
(define (sum-square x y)
  (+ (square x) (square y)))
(define (square-2of3 a b c)
  (cond ((and (>= a b) (>= b c)) (sum-square a b))
        ((and (>= a b) (> c b)) (sum-square a c))
        (else (sum-square b c))))
```

```
<<EX1-3>>
<<try-these>>
 (try-these square-2of3 '(7 5 3)
                        '(7 3 5)
                        '(3 5 7))
```

(7 5 3)	74
(7 3 5)	74
(3 5 7)	74

2.10 Exercise 1.4

2.10.1 Question

> Observe that our model of evaluation allows for combinations whose operators are compound expressions. Use this observation to describe the behavior of the following procedure:

```scheme
(define (a-plus-abs-b a b)
  ((if (> b 0) + -) a b))
```

2.10.2 Answer

This code accepts the variables `a` and `b`, and if `b` is positive, it adds `a` and `b`. However, if `b` is zero or negative, it subtracts them. This decision is made by using the `+` and `-` procedures as the results of an if expression, and then evaluating according to the results of that expression. This is in contrast to a language like Python, which would do something like this:

```python
if b > 0: a + b
else: a - b
```

2.11 Exercise 1.5

2.11.1 Question

> Ben Bitdiddle has invented a test to determine whether the interpreter he is faced with is using applicative-order evaluation or normal-order evaluation. He defines the following two procedures:

```scheme
(define (p) (p))

(define (test x y)
  (if (= x 0)
      0
      y))
```

> Then he evaluates the expression:

```scheme
(test 0 (p))
```

> What behavior will Ben observe with an interpreter that uses applicative-order evaluation? What behavior will he observe with an interpreter that uses normal-order evaluation? Explain your answer. (Assume

that the evaluation rule for the special form if is the same whether the interpreter is using normal or applicative order: The predicate expression is evaluated first, and the result determines whether to evaluate the consequent or the alternative expression.)

2.11.2 Answer

In either type of language, `(define (p) (p))` is an infinite loop. However, a normal-order language will encounter the special form, return `0`, and never evaluate `(p)`. An applicative-order language evaluates the arguments to `(test 0 (p))`, thus triggering the infinite loop.

2.12 1.1.7: Example: Square Roots by Newton's Method

Functions in the formal mathematical sense are **declarative knowledge**, while procedures like in computer science are **imperative knowledge**.

Notice that the elements of the language that have been introduced so far are sufficient for writing any purely numerical program, despite not having introduced any looping constructs like `FOR` loops.

2.13 1.1.8: Procedures as Black-Box Abstractions

Notice how the `sqrt` procedure is divided into other procedures, which mirror the division of the square root problem into sub problems.

A procedure should accomplish an identifiable task, and be ready to be used as a module in defining other procedures. This lets the programmer know how to use the procedure while not needing to know the details of how it works.

Suppressing these details are particularly helpful:

Local names. A procedure user shouldn't need to know a procedure's choices of variable names. A formal parameter of a procedure whose name is irrelevant is called a **bound variable**. A procedure definition **binds** its parameters. A **free variable** isn't bound. The set of expressions in which a binding defines a name is the **scope** of that name.

Internal definitions and block structure. By nesting relevant definitions inside other procedures, you hide them from the global namespace. This nesting is called **block structure**. Nesting these definitions also allows relevant variables to be shared across procedures, which is called **lexical scoping**.

2.14 Exercise 1.6

2.14.1 Text code

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

```scheme
      (- x)
      x))
```

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

```scheme
<<average>>
(define (improve guess x)
  (average guess (/ x guess)))

<<square>>
<<abs>>
(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))
```

2.14.2 Question

Alyssa P. Hacker doesn't see why `if` needs to be provided as a special form. "Why can't I just define it as an ordinary procedure in terms of cond?" she asks. Alyssa's friend Eva Lu Ator claims this can indeed be done, and she defines a new version of `if`:

```scheme
(define (new-if predicate
                then-clause
                else-clause)
  (cond (predicate then-clause)
        (else else-clause)))
```

Eva demonstrates the program for Alyssa:

```scheme
(new-if (= 2 3) 0 5)
;; => 5

```

```
(new-if (= 1 1) 0 5)
;; => 0
```

Delighted, Alyssa uses new-if to rewrite the square-root program:

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

What happens when Alyssa attempts to use this to compute square roots? Explain.

2.14.3 Answer

Using Alyssa's `new-if` leads to an infinite loop because the recursive call to `sqrt-iter` is evaluated before the actual call to `new-if`. This is because `if` and `cond` are special forms that change the way evaluation is handled; whichever branch is chosen leaves the other branches unevaluated.

2.15 Exercise 1.7

2.15.1 Text

```
(define (mean-square x y)
  (average (square x) (square y)))
```

2.15.2 Question

The `good-enough?` test used in computing square roots will not be very effective for finding the square roots of very small numbers. Also, in real computers, arithmetic operations are almost always performed with limited precision. This makes our test inadequate for very large numbers. Explain these statements, with examples showing how the test fails for small and large numbers. An alternative strategy for implementing `good-enough?` is to watch how guess changes from one iteration to the next and to stop when the change is a very small fraction of the guess. Design a square-root procedure that uses this kind of end test. Does this work better for small and large numbers?

2.15.3 Diary

1. Solving My original answer was this, which compares the previous iteration until the new and old are within an arbitrary dx.

```
<<txt-sqrt>>
(define (inferior-good-enough? guess lastguess)
  (<=
   (abs (-
         (/ lastguess guess)
         1))
   0.0000000000001)) ; dx
(define (new-sqrt-iter guess x lastguess) ;; Memory of previous value
  (if (inferior-good-enough? guess lastguess)
      guess
      (new-sqrt-iter (improve guess x) x guess)))
(define (new-sqrt x)
  (new-sqrt-iter 1.0 x 0))
```

This solution can correctly find small and large numbers:

```
<<inferior-good-enough>>
(new-sqrt 10000000000000)
```

```
3162277.6601683795
```

```
<<try-these>>
<<inferior-good-enough>>
(try-these new-sqrt '(0.01 0.0001 0.000001 0.00000001 0.0000000001))
```

0.01	0.1
0.0001	0.01
1e-06	0.001
1e-08	9.999999999999999e-05
1e-10	9.999999999999999e-06

However, I found this solution online that isn't just simpler but automatically reaches the precision limit of the system:

```
<<txt-sqrt>>
(define (best-good-enough? guess x)
   (= (improve guess x) guess))
```

2. Imroving (sqrt) by avoiding extra (improve) call

   (a) Non-optimized

```scheme
(use-modules (ice-9 format))
(load ”../mattbench.scm”)
(define (average x y)
  (/ (+ x y) 2))
(define (improve guess x)
  (average guess (/ x guess)))
(define (good-enough? guess x)
   (= (improve guess x) guess)) ;; improve call 1
(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x) x))) ;; call 2
(define (sqrt x)
  (sqrt-iter 1.0 x))
(newline)
(display (mattbench (λ() (sqrt 69420)) 400000000))
(newline)
;; 4731.30 <- Benchmark results
```

(b) Optimized

```scheme
(use-modules (ice-9 format))
(load ”../mattbench.scm”)
(define (average x y)
  (/ (+ x y) 2))
(define (improve guess x)
  (average guess (/ x guess)))
(define (good-enough? guess nextguess x)
  (= nextguess guess))
(define (sqrt-iter guess x)
  (let ((nextguess (improve guess x)))
    (if (good-enough? guess nextguess x)
        guess
        (sqrt-iter nextguess x))))
(define (sqrt x)
  (sqrt-iter 1.0 x))
(newline)
(display (mattbench (λ() (sqrt 69420)) 400000000))
(newline)
```

(c) Benchmark results

Unoptimized	4731.30
Optimized	2518.44

2.15.4 Answer

The current method has decreasing accuracy with smaller numbers. Notice the steady divergence from correct answers here (should be decreasing powers of 0.1):

```
<<txt-sqrt>>
<<try-these>>
(try-these sqrt 0.01 0.0001 0.000001 0.00000001 0.0000000001)
```

0.01	0.10032578510960605
0.0001	0.03230844833048122
1e-06	0.031260655525445276
1e-08	0.03125010656242753
1e-10	0.03125000106562499

And for larger numbers, an infinite loop will eventually be reached. 10^{12} can resolve, but 10^{13} cannot.

```
<<txt-sqrt>>
(sqrt 1000000000000)
```

```
1000000.0
```

So, my definition of `sqrt`:

```
<<average>>
(define (improve guess x)
  (average guess (/ x guess)))
(define (good-enough? guess x)
   (= (improve guess x) guess))
(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))
```

```
<<try-these>>
<<sqrt>>
(try-these sqrt '(0.01 0.0001 0.000001 0.00000001 0.0000000001))
```

0.01	0.1
0.0001	0.01
1e-06	0.001
1e-08	9.999999999999999e-05
1e-10	9.999999999999999e-06

2.16 Exercise 1.8

2.16.1 Question

> Newton's method for cube roots is based on the fact that if y is an approximation to the cube root of x, then a better approximation is given by the value:
>
> $$\frac{\frac{x}{y^2} + 2y}{3} \tag{1}$$
>
> Use this formula to implement a cube-root procedure analogous to the square-root procedure. (In 1.3.4 we will see how to implement Newton's method in general as an abstraction of these square-root and cube-root procedures.)

2.16.2 Diary

My first attempt works, but needs an arbitrary limit to stop infinite loops:

```scheme
<<square>>
<<try-these>>
(define (cb-good-enough? guess x)
  (= (cb-improve guess x) guess))
(define (cb-improve guess x)
  (/
   (+
    (/ x (square guess))
    (* guess 2))
   3))
(define (cbrt-iter guess x counter)
  (if (or (cb-good-enough? guess x) (> counter 100))
      guess
      (begin
        (cbrt-iter (cb-improve guess x) x (+ 1 counter)))))
(define (cbrt x)
  (cbrt-iter 1.0 x 0))

(try-these cbrt 7 32 56 100)
```

7	1.912931182772389
32	3.174802103936399
56	3.825862365544778
100	4.641588833612779

However, this will hang on an infinite loop when trying to run `(cbrt 100)`. I speculate it's a floating point precision issue with the "improve" algorithm. So to avoid it I'll just keep track of the last guess and stop improving when there's no more change occurring. Also while researching I discovered that (again due to floating point) `(cbrt -2)` loops forever unless you initialize your guess with a slightly different value, so let's do 1.1 instead.

2.16.3 Answer

```
<<square>>
(define (cb-good-enough? nextguess guess lastguess x)
  (or (= nextguess guess)
      (= nextguess lastguess)))
(define (cb-improve guess x)
  (/
   (+
    (/ x (square guess))
    (* guess 2))
   3))
(define (cbrt-iter guess lastguess x)
  (define nextguess (cb-improve guess x))
  (if (cb-good-enough? nextguess guess lastguess x)
      nextguess
      (cbrt-iter nextguess guess x)))
(define (cbrt x)
  (cbrt-iter 1.1 9999 x))
```

```
<<cbrt>>
<<try-these>>
(try-these cbrt 7 32 56 100 -2)
```

7	1.912931182772389
32	3.174802103936399
56	3.825862365544778
100	4.641588833612779
-2	-1.2599210498948732

2.17 1.2: Procedures and the Processes They Generate

Procedures define the **local evolution** of processes. We would like to be able to make statements about the **global** behavior of a process.

2.18 1.2.1: Linear Recursion and Iteration

Consider these two procedures for obtaining factorials:

```
(define (factorial-recursion n)
  (if (= n 1)
      1
      (* n
         (factorial-recursion (- n 1)))))

(define (factorial-iteration n)
  (define (fact-iter product counter max-count)
      (if (> counter max-count)
          product
          (fact-iter
                    (* counter product)
                    (+ counter 1)
                    max-count)))

  (fact-iter 1 1 n))
```

These two procedures reach the same answers, but form very different processes. The `factorial-recursion` version takes more computational **time** and **space** to evaluate, by building up a chain of deferred operations. This is a **recursive process**. As the number of steps needed to operate, and the amount of info needed to keep track of these operations, both grow linearly with n, this is a **linear recursive process**.

The second version forms an **iterative process**. Its state can be summarized with a fixed number of state variables. The number of steps required grow linearly with n, so this is a **linear iterative process**.

recursive procedure is a procedure whose definition refers to itself.

recursive process is a process that evolves recursively.

So `fact-iter` is a recursive *procedure* that generates an iterative *process*.

Many implementations of programming languages interpret all recursive procedures in a way that consume memory that grows with the number of procedure calls, even when the process is essentially iterative. These languages instead use looping constructs such as `do`, `repeat`, `for`, etc. Implementations that execute iterative processes in constant space, even if the procedure is recursive, are **tail-recursive**.

2.19 Exercise 1.9

2.19.1 Question

> Each of the following two procedures defines a method for adding two positive integers in terms of the procedures inc, which increments its argument by 1, and dec, which decrements its argument by 1.

```scheme
(define (+ a b)
  (if (= a 0)
      b
      (inc (+ (dec a) b))))

(define (+ a b)
  (if (= a 0)
      b
      (+ (dec a) (inc b))))
```

Using the substitution model, illustrate the process generated by each procedure in evaluating `(+ 4 5)`. Are these processes iterative or recursive?

2.19.2 Answer

The first procedure is recursive, while the second is iterative though tail-recursion.

1. recursive procedure

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

2. iterative procedure

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

2.20 Exercise 1.10

2.20.1 Question

The following procedure computes a mathematical function called Ackermann's function.

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

What are the values of the following expressions?

```scheme
(A 1 10)
(A 2 4)
(A 3 3)
```

(1 10)	1024
(2 4)	65536
(3 3)	65536

```scheme
<<ackermann>>
(define (f n) (A 0 n))
(define (g n) (A 1 n))
(define (h n) (A 2 n))
(define (k n) (* 5 n n))
```

Give concise mathematical definitions for the functions computed by the procedures `f`, `g`, and `h` for positive integer values of n. For example, `(k n)` computes $5n^2$.

2.20.2 Answer

1. `f`

   ```scheme
   <<try-these>>
   <<EX1-10-defs>>
   (try-these f 1 2 3 10 15 20)
   ```

1	2
2	4
3	6
10	20
15	30
20	40

   $$f(n) = 2n$$

2. `g`

```
<<try-these>>
<<EX1-10-defs>>
(try-these g 1 2 3 4 5 6 7 8)
```

1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256

$$g(n) = 2^n$$

3. `h`

```
<<try-these>>
<<EX1-10-defs>>
(try-these h 1 2 3 4)
```

1	2
2	4
3	16
4	65536

It took a while to figure this one out, just because I didn't know the term. This is repeated exponentiation. This operation is to exponentiation, what exponentiation is to multiplication. It's called either *tetration* or *hyper-4* and has no formal notation, but two common ways would be these:

$$h(n) = 2 \uparrow\uparrow n$$

$$h(n) = {}^{n}2$$

2.21 1.2.2: Tree Recursion

Consider a recursive procedure for computing Fibonacci numbers:

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
```

```
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))
```

The resulting process splits into two with every iteration, creating a tree of computations, many of which are duplicates of previous computations. This kind of pattern is called **tree-recursion**. However, this one is quite inefficient. The time and space required grows exponentially with the number of iterations requested.

Instead, it makes much more sense to start from `Fib(1) ~ 1` and `Fib(0) ~ 0` and iterate upwards to the desired value. This only requires a linear number of steps relative to the input.

```
(define (fib n)
  (fib-iter 1 0 n))
(define (fib-iter a b count)
  (if (= count 0) b (fib-iter (+ a b) a (- count 1))))
```

However, notice that the inefficient tree-recursive version is a fairly straightforward translation of the Fibonacci sequence's definition, while the iterative version required redefining the process as an iteration with three variables.

2.21.1 Example: Counting change

I should come back and try to make the "better algorithm" suggested.

2.22 Exercise 1.11

2.22.1 Question

A function f is defined by the rule that:

$f(n) = n$ if $n < 3$

and

$f(n) = f(n-1) + 2f(n-2) + 3f(n-3)$ if $n \geq 3$

Write a procedure that computes f by means of a recursive process. Write a procedure that computes f by means of an iterative process.

2.22.2 Answer

1. Recursive

```
(define (fr n)
  (if (< n 3)
      n
      (+      (fr (- n 1))
         (* 2 (fr (- n 2)))
```

```
        (* 3 (fr (- n 3))))))
```

```
<<try-these>>
<<EX1-11-fr>>
(try-these fr 1 3 5 10)
```

1	1
3	4
5	25
10	1892

2. Iterative

   (a) Attempt 1

```
;; This seems like it could be better
(define (fi n)
  (define (formula l)
    (let ((a (car l))
          (b (cadr l))
          (c (caddr l)))
      (+ a
         (* 2 b)
         (* 3 c))))
  (define (iter l i)
    (if (= i n)
        (car l)
        (iter (cons (formula l) l)
              (+ 1 i))))
  (if (< n 3)
      n
      (iter '(2 1 0) 2)))
```

```
<<try-these>>
<<EX1-11-fi>>
(try-these fi 1 3 5 10)
```

1	1
3	4
5	25
10	1892

It works but it seems wasteful.

   (b) Attempt 2

```scheme
(define (fi2 n)
  (define (formula a b c)
      (+ a
         (* 2 b)
         (* 3 c)))
  (define (iter a b c i)
    (if (= i n)
        a
        (iter (formula a b c)
              a
              b
              (+ 1 i))))
  (if (< n 3)
      n
      (iter 2 1 0 2)))
```

```scheme
<<try-these>>
<<EX1-11-fi2>>
(try-these fi2 1 3 5 10)
```

1	1
3	4
5	25
10	1892

I like that better.

2.23 Exercise 1.12

2.23.1 Question

The following pattern of numbers is called Pascal's triangle.

Pretend there's a Pascal's triangle here.

The numbers at the edge of the triangle are all 1, and each number inside the triangle is the sum of the two numbers above it. Write a procedure that computes elements of Pascal's triangle by means of a recursive process.

2.23.2 Answer

I guess I'll rotate the triangle 45 degrees to make it the top-left corner of an infinite spreadsheet.

```scheme
(define (pascal x y)
  (if (or (= x 0)
          (= y 0))
      1
      (+ (pascal (- x 1) y)
         (pascal x (- y 1)))))
```

```scheme
<<try-these>>
<<pascal-rec>>
(let ((l (iota 8)))
  (map (λ (row)
         (map (λ (xy)
                (apply pascal xy))
              row))
       (map (λ (x)
              (map (λ (y)
                     (list x y))
                   l))
            l)))
```

1	1	1	1	1	1	1	1
1	2	3	4	5	6	7	8
1	3	6	10	15	21	28	36
1	4	10	20	35	56	84	120
1	5	15	35	70	126	210	330
1	6	21	56	126	252	462	792
1	7	28	84	210	462	924	1716
1	8	36	120	330	792	1716	3432

The test code was much harder to write than the actual solution.

2.24 Exercise 1.13

2.24.1 Question

Prove that Fib(n) is the closest integer to $\frac{{}^n}{\sqrt{5}}$ where Phi is $\frac{1+\sqrt{5}}{2}$. Hint: let $= \frac{1-\sqrt{5}}{2}$. Use induction and the definition of the Fibonacci numbers to prove that

$$\text{Fib}(n) = \frac{{}^n - {}^n}{\sqrt{5}}$$

2.24.2 Answer

I don't know how to write a proof yet, but I can make functions to demonstrate it.

1. Fibonacci number generator

```scheme
(define (fib-iter n)
  (define (iter i a b)
    (if (= i n)
        b
    (iter (+ i 1)
          b
          (+ a b))))
  (if (<= n 2)
      1
      (iter 2 1 1)))
```

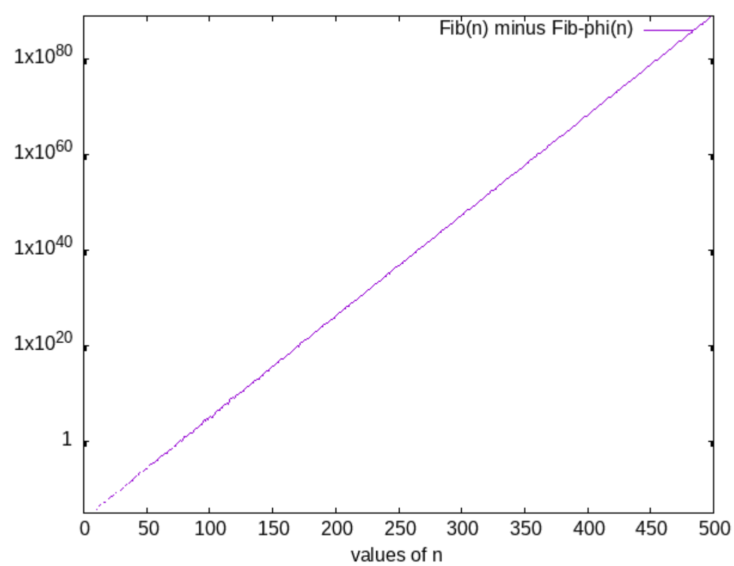
2. Various algorithms relating to the question

```scheme
<<sqrt>>
(define sqrt5
  (sqrt 5))
(define phi
  (/ (+ 1 sqrt5) 2))
(define upsilon
  (/ (- 1 sqrt5) 2))
(define (fib-phi n)
  (/ (- (expt phi n)
        (expt upsilon n))
     sqrt5))
```

```scheme
(use-srfis '(1))
<<fib-iter>>
<<fib-phi>>
<<try-these>>

(let* ((vals (drop (iota 21) 10))
       (fibs (map fib-iter vals))
       (approx (map fib-phi vals)))
  (zip vals fibs approx))
```

10	55	54.99999999999999
11	89	89.0
12	144	143.99999999999997
13	233	232.99999999999994
14	377	377.00000000000006
15	610	610.0
16	987	986.9999999999998
17	1597	1596.9999999999998
18	2584	2584.0
19	4181	4181.0
20	6765	6764.999999999999

You can see they follow closely. Graphing the differences, it's just an exponential curve at very low values, presumably following the exponential increase of the Fibonacci sequence itself.



2.25 1.2.3: Orders of Growth

An **order of growth** gives you a gross measure of the resources required by a process as its inputs grow larger.

Let n be a parameter for the size of a problem, and $R(n)$ be the amount of resources required for size n. $R(n)$ has order of growth $\Theta(f(n))$

For example:

$\Theta(1)$ is constant, not growing regardless of input size.

$\Theta(n)$ is growth 1-to-1 proportional to the input size.

Some algorithms we've already seen:

Linear recursive is time and space $\Theta(n)$

Iterative is time $\Theta(n)$ space $\Theta(1)$

Tree-recursive means in general, time is proportional to the number of nodes, space is proportional to the depth of the tree. In the Fibonacci algorithm example, $\Theta(n)$ and time $\Theta(\Upsilon^n)$ where Υ is the golden ratio $\frac{1+\sqrt{5}}{2}$

Orders of growth are very crude descriptions of process behaviors, but they are useful in indicating how a process will change with the size of the problem.

2.26 Exercise 1.14

Below is the default version of the count-change function. I'll be aggressively modifying it in order to get a graph out of it.

```scheme
(define (count-change amount)
  (cc amount 5))

(define (cc amount kinds-of-coins)
  (cond ((= amount 0) 1)
        ((or (< amount 0)
             (= kinds-of-coins 0))
         0)
        (else
         (+ (cc amount (- kinds-of-coins 1))
            (cc (- amount (first-denomination
                           kinds-of-coins))
                kinds-of-coins)))))

(define (first-denomination kinds-of-coins)
  (cond ((= kinds-of-coins 1) 1)
        ((= kinds-of-coins 2) 5)
        ((= kinds-of-coins 3) 10)
        ((= kinds-of-coins 4) 25)
        ((= kinds-of-coins 5) 50)))
```

2.26.1 Question A

Draw the tree illustrating the process generated by the count-change procedure of 1.2.2 in making change for 11 cents.

2.26.2 Answer

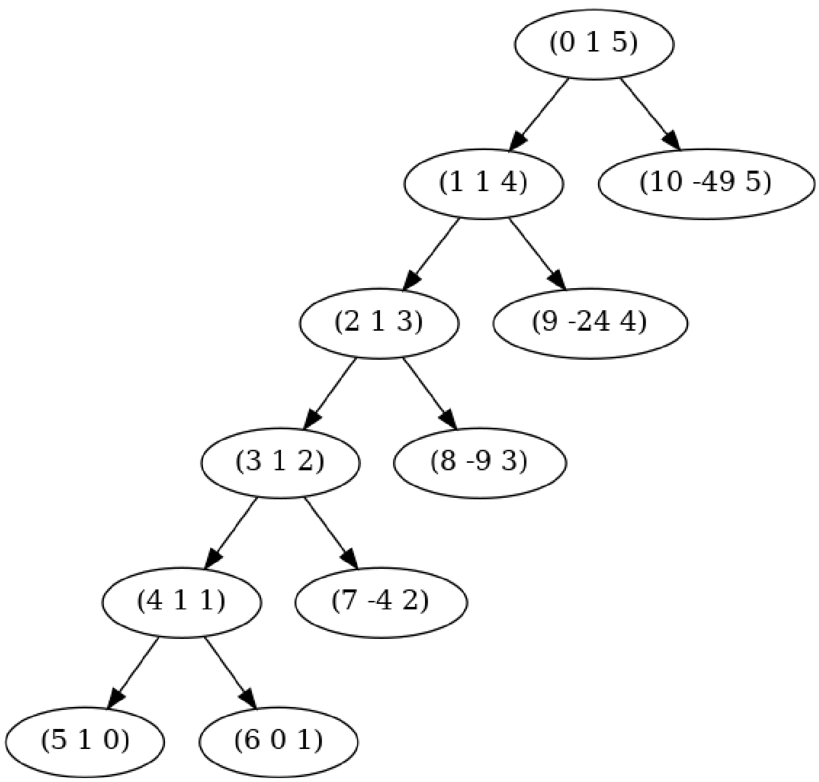
I want to generate this graph algorithmically.

```scheme
;; cursed global
(define bubblecounter 0)
;; Returns # of ways change can be made
;; "Helper" for (cc)
(define (count-change amount)
  (display "digraph {\n") ;; start graph
  (cc amount 5 0)
  (display "}\n") ;; end graph
  (set! bubblecounter 0))

;; GraphViz output
;; Derivative: https://stackoverflow.com/a/14806144
(define (cc amount kinds-of-coins oldbubble)
  (let ((recur (lambda (new-amount new-kinds)
               (begin
                 (display "\"") ;; Source bubble
                 (display `(,oldbubble ,amount ,kinds-of-coins))
                 (display "\"")
                 (display " -> ") ;; arrow pointing from parent to child
                 (display "\"") ;; child bubble
                 (display `(,bubblecounter ,new-amount ,new-kinds))
                 (display "\"")
                 (display "\n")
                 (cc new-amount new-kinds bubblecounter)))))
    (set! bubblecounter (+ bubblecounter 1))
    (cond ((= amount 0) 1)
          ((or (< amount 0) (= kinds-of-coins 0)) 0)
          (else (+
                 (recur amount (- kinds-of-coins 1))
                 (recur (- amount
                           (first-denomination kinds-of-coins))
                        kinds-of-coins))))))

(define (first-denomination kinds-of-coins)
  (cond ((= kinds-of-coins 1) 1)
        ((= kinds-of-coins 2) 5)
        ((= kinds-of-coins 3) 10)
        ((= kinds-of-coins 4) 25)
        ((= kinds-of-coins 5) 50)))
```

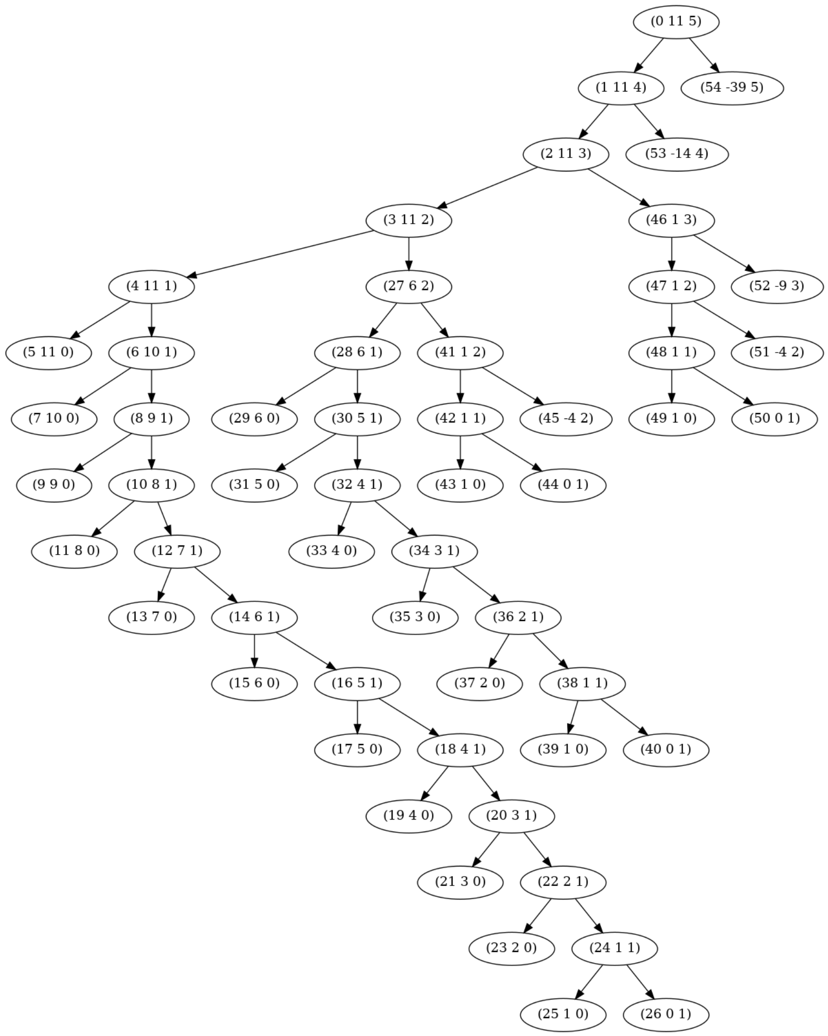
I'm not going to include the full printout of the `(count-change 11)`, here's an example of what this looks like via `1`.

```
<<count-change-graphviz>>
(count-change 1)
```

```
digraph {
"(0 1 5)" -> "(1 1 4)"
"(1 1 4)" -> "(2 1 3)"
"(2 1 3)" -> "(3 1 2)"
"(3 1 2)" -> "(4 1 1)"
"(4 1 1)" -> "(5 1 0)"
"(4 1 1)" -> "(6 0 1)"
"(3 1 2)" -> "(7 -4 2)"
"(2 1 3)" -> "(8 -9 3)"
"(1 1 4)" -> "(9 -24 4)"
"(0 1 5)" -> "(10 -49 5)"
}
```



So, the graph of `(count-change 11)` is:



2.26.3 Question B

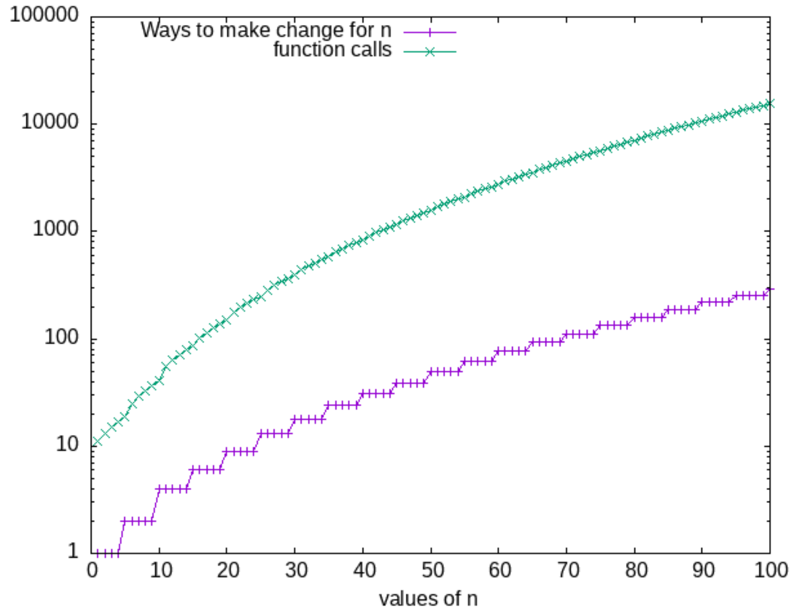
> What are the orders of growth of the space and number of steps used by this process as the amount to be changed increases?

2.26.4 Answer B

Let's look at this via the number of function calls needed for value `n`. Instead of returning an integer, I'll return a pair where `car` is the number of ways to count change, and `cdr` is the number of function calls that have occurred down that branch of the tree.

```scheme
(define (count-calls amount)
  (cc-calls amount 5))

(define (cc-calls amount kinds-of-coins)
  (cond ((= amount 0) '(1 . 1))
        ((or (< amount 0)
             (= kinds-of-coins 0))
         '(0 . 1))
        (else
         (let ((a (cc-calls amount (- kinds-of-coins 1)))
               (b (cc-calls (- amount (first-denomination
                                       kinds-of-coins))
                            kinds-of-coins)))
           (cons (+ (car a)
                    (car b))
                 (+ 1
                    (cdr a)
                    (cdr b)))))))

(define (first-denomination kinds-of-coins)
  (cond ((= kinds-of-coins 1) 1)
        ((= kinds-of-coins 2) 5)
        ((= kinds-of-coins 3) 10)
        ((= kinds-of-coins 4) 25)
        ((= kinds-of-coins 5) 50)))
```

```scheme
(use-srfis '(1))
<<cc-calls>>
(let* ((vals (drop (iota 101) 1))
       (mine (map count-calls vals)))
  (zip vals (map car mine) (map cdr mine)))
```



I believe the space to be $\Theta(n + d)$ as the function calls count down the denominations before counting down the change. However I notice most answers describe $\Theta(n)$ instead, maybe I'm being overly pedantic and getting the wrong answer.

My issues came finding the time. The book describes the meaning and properties of Θ notation in Section 1.2.3. However, my lack of formal math education made realizing the significance of this passage difficult. For one, I didn't understand that $k_1 f(n) \leq R(n) \leq k_2 f(n)$ means "you can find the Θ by proving that a graph of the algorithm's resource usage is bounded by two identical functions multiplied by constants." So, the graph of resource usage for an algorithm with $\Theta(n^2)$ will by bounded by lines of $n^2 \times someconstant$, the top boundary's constant being larger than the small boundary. These are arbitrarily chosen constants, you're just proving that the function behaves the way you think it does.

Overall, finding the Θ and Ω and O notations (they are all different btw!) is about aggressively simplifying to make a very general statement about the behavior of the algorithm.

I could tell that a "correct" way to find the Θ would be to make a formula which describes the algorithm's function calls for given input and denominations. This is one of the biggest time sinks, although I had a lot of fun and learned a lot. In the end, with some help from Jach in a Lisp Discord, I had the following formula:

$$\sum_{i=1}^{ceil(n/val(d))} T(n - val(d) * i, d)$$

But I wasn't sure where to go from here. The graphs let me see some interesting trends, though I didn't get any closer to an answer in the process.

By reading on other websites, I knew that you could find Θ by obtaining a formula for $R(n)$ and removing constants to end up with a term of interest. For example, if your algorithm's resource usage is $\frac{n^2+7n}{5}$, this demonstrates $\Theta(n^2)$. So I know a formula **without** a $\sum$ would give me the answer I wanted. It didn't occur to me that it might be possible to use calculus to remove the $\sum$ from the equation. At this point I knew I was stuck and decided to look up a guide.

After seeing a few solutions that I found somewhat confusing, I landed on this awesome article from Codology.net. They show how you can remove the summation, and proposed this equation for count-change with 5 denominations:

$$T(n,5) = \frac{n}{50} + 1 + \sum_{i=0}^{n/50} T(n-50i,1)$$

Which, when expanded and simplified, demonstrates $\Theta(n^5)$ for 5 denominations.

Overall I'm relieved that I wasn't entirely off, given I haven't done math work like this since college. It's inspired me to restart my remedial math courses, I don't think I really grasped the nature of math as a tool of empowerment until now.

2.27 Exercise 1.15

2.27.1 Question A

> The sine of an angle (specified in radians) can be computed by making use of the approximation $\sin x\ x$ if x is sufficiently small, and the trigonometric identity $\sin x = 3\sin\frac{x}{3} - 4\sin^3\frac{x}{3}$ to reduce the size of the argument of sin. (For purposes of this exercise an angle is considered "sufficiently small" if its magnitude is not greater than 0.1 radians.) These ideas are incorporated in the following procedures:

```
(define (cube x) (* x x x))
(define (p x) (- (* 3 x) (* 4 (cube x))))
(define (sine angle)
  (if (not (> (abs angle) 0.1))
      angle
      (p (sine (/ angle 3.0)))))
```

> How many times is the procedure `p` applied when `(sine 12.15)` is evaluated?

2.27.2 Answer A

Let's find out!

```
(define (cube x) (* x x x))
(define (p x) (- (* 3 x) (* 4 (cube x))))
(define (sine angle)
  (if (not (> (abs angle) 0.1))
      (cons angle 0)
      (let ((x (sine (/ angle 3.0))))
        (cons (p (car x)) (+ 1 (cdr x))))))
```

```
<<1-15-p-measure>>
(let ((xy (sine 12.15)))
  (list (car xy) (cdr xy)))
```

-0.39980345741334	5

`p` is evaluated 5 times.

2.27.3 Question B

> What is the order of growth in space and number of steps (as a function of `a`) used by the process generated by the sine procedure when `(sine a)` is evaluated?

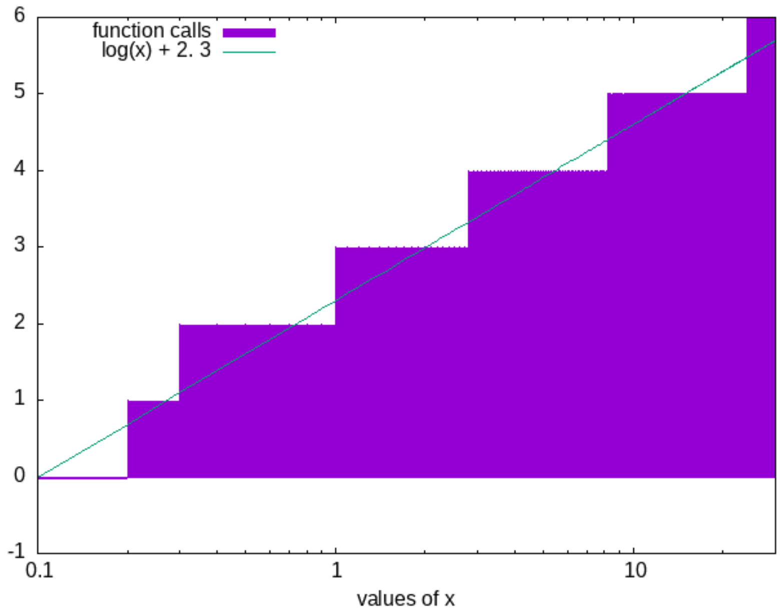
2.27.4 Answer B

```
(use-srfis '(1))
<<1-15-p-measure>>
(let* ((vals (iota 300 0.1 0.1))
       (sines (map (λ (i)
                     (cdr (sine i)))
                   vals)))
  (zip vals sines))
```

```
(use-srfis '(1))
<<1-15-p-measure>>
(let* ((vals (iota 10 0.1 0.1))
       (sines (map (λ (i)
                     (cdr (sine i)))
                   vals)))
  (zip vals sines))
```

Example output:

0.1	0
0.2	1
0.30000000000000004	2
0.4	2
0.5	2
0.6	2
0.7000000000000001	2
0.8	2
0.9	2
1.0	3



This graph shows that the number of times `sine` will be called is logarithmic.

- 0.1 to 0.2 are divided once
- 0.3 to 0.8 are divided twice
- 0.9 to 2.6 are divided three times
- 2.7 to 8 are divided four times
- 8.5 to 23.8 are divided five times

Given that the calls to `p` get stacked recursively, like this:

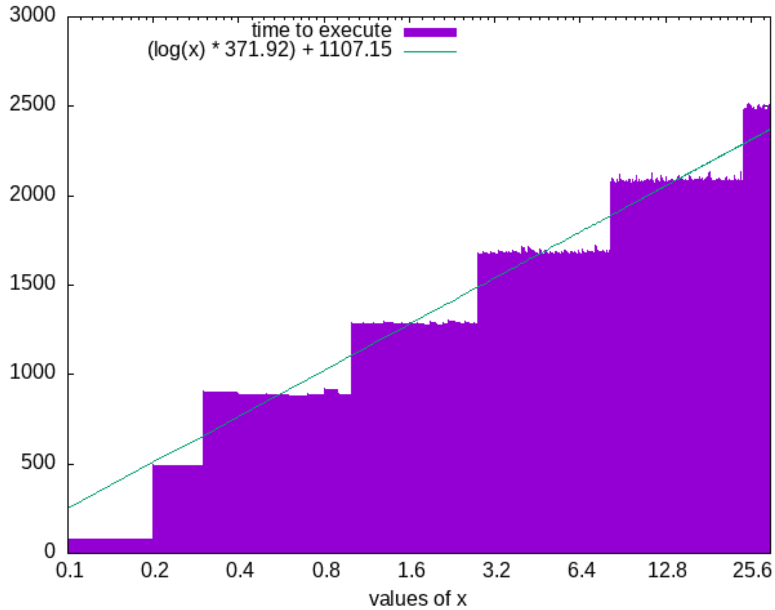
```
(sine 12.15)
(p (sine 4.05))
(p (p (sine 1.35)))
```

```
(p (p (p (sine 0.45))))
(p (p (p (p (sine 0.15)))))
(p (p (p (p (p (sine 0.05))))))
(p (p (p (p (p 0.05)))))
(p (p (p (p 0.14950000000000002))))
(p (p (p 0.43513455050000005)))
(p (p 0.9758465331678772))
(p -0.7895631144708228)
-0.39980345741334
```

So I argue the space and time is $\Theta(\log(n))$
We can also prove this for the time by benchmarking the function:

```
;; This execution takes too long for org-mode, so I'm doing it
;; externally and importing the results
(use-srfis '(1))
(use-modules (ice-9 format))
(load ”../../mattbench.scm”)
<<1-15-deps>>
(let* ((vals (iota 300 0.1 0.1))
       (times (map (λ (i)
                     (mattbench (λ () (sine i)) 1000000))
                   vals)))
  (with-output-to-file ”sine-bench.dat” (λ ()
     (map (λ (x y)
           (format #t ”~s~/~s~%” x y))
         vals times))))
```

```
reset # helps with various issues in execution
set xtics 0.5
set xlabel 'values of x'
set logscale x
set key top left
set style fill solid 1.00 border
#set style function fillsteps below

f(x) = (log(x) * a) + b
fit f(x) 'Ex15/sine-bench.dat' using 1:2 via a,b

plot 'Ex15/sine-bench.dat' using 1:2 with fillsteps title 'time to execute', \
     'Ex15/sine-bench.dat' using 1:(f($1)) with lines title sprintf('(log(x) * %.2f) + %.2f', a, b)
```



1. 1.2.4 Exponentiation Considering a few ways to compute the exponential of a given number.

```
(define (expt b n)
  (expt-iter b n 1))
(define (expt-iter b counter product)
  (if (= counter 0)
      product
      (expt-iter b (- counter 1) (* b product))))
```

This iterative procedure is essentially equivalent to:

$$b^8 = b \cdot (b \cdot (b \cdot (b \cdot (b \cdot (b \cdot (b \cdot b))))))$$

But note it could be approached faster with squaring:

$$b^2 = b \cdot b$$
$$b^4 = b^2 \cdot b^2$$
$$b^8 = b^4 \cdot b^4$$

2.28 Exercise 1.16

2.28.1 Text

```scheme
(define (expt-rec b n)
  (if (= n 0)
      1
      (* b (expt-rec b (- n 1)))))

(define (expt-iter b n)
  (define (iter counter product)
    (if (= counter 0)
        product
        (iter (- counter 1)
              (* b product))))
  (iter n 1))

(define (fast-expt b n)
  (cond ((= n 0)
         1)
        ((even? n)
         (square (fast-expt b (/ n 2))))
        (else
         (* b (fast-expt b (- n 1))))))
```

2.28.2 Question

> Design a procedure that evolves an iterative exponentiation process that uses successive squaring and uses a logarithmic number of steps, as does fast-expt. (Hint: Using the observation that $(b^{n/2})^2 = (b^2)^{n/2}$, keep, along with the exponent n and the base b, an additional state variable a , and define the state transformation in such a way that the product ab^n is unchanged from state to state. At the beginning of the process a is taken to be 1, and the answer is given by the value of a at the end of the process. In general, the technique of defining an *invariant quantity* that remains unchanged from state to state is a powerful way to think about the design of iterative algorithms.)

2.28.3 Diary

First I made this program which tries to use a false equivalence:

$$ab^2 = (a+1)b^{n-1}$$

```scheme
<<square>>
(define (fast-expt-iter b n)
  (define (iter b n a)
```

```
    (format #t "~&|~s~/~/|~s~/~/|~s|~%" b n a)
    (cond ((= n 1) (begin (format #t "~&|~s~/~/|~s~/~/|~s|~%" (* b a) 1 1)
                          (* b a)))
          ((even? n) (iter (square b)
                           (/ n 2)
                           a))
          (else (iter b (- n 1) (+ a 1)))))
  (format #t "|~a~/|~a~/|~a|~%" "base" "power" "variable")
  (format #t "~&|--|--|--|~%")
  (iter b n 1))
```

```
<<fast-expt-iter-fail1>>
<<try-these>>
(fast-expt-iter 2 6)
```

Here's what the internal state looks like during 2^6 (correct answer is 64):

base	power	variable
2	6	1
4	3	1
4	2	2
16	1	2
32	1	1

2.28.4 Answer

There are two key transforms to a faster algorithm. The first was already shown in the text:

$$ab^n \to a(b^2)^{n/2}$$

The second which I needed to deduce was this:

$$ab^n \to ((a \times b) \times b)^{n-1}$$

The solution essentially follows this logic:

- initialize a to 1
- If n is 1, return $b * a$
- else if n is even, halve n, square b, and iterate
- else n is odd, so subtract 1 from n and $a \to a \times b$

```scheme
<<square>>
(define (fast-expt-iter b n)
  (define (iter b n a)
    (cond ((= n 1) (* b a))
          ((even? n) (iter (square b)
                           (/ n 2)
                           a))
          (else (iter b (- n 1) (* b a)))))
  (iter b n 1))
```

```scheme
<<fast-expt-iter>>
<<try-these>>
(try-these (λ(x) (fast-expt-iter 3 x)) (cdr (iota 11)))
```

1	3
2	9
3	27
4	81
5	243
6	729
7	2187
8	6561
9	19683
10	59049

2.29 Exercise 1.17

2.29.1 Question

The exponentiation algorithms in this section are based on performing exponentiation by means of repeated multiplication. In a similar way, one can perform integer multiplication by means of repeated addition. The following multiplication procedure (in which it is assumed that our language can only add, not multiply) is analogous to the expt procedure:

```scheme
(define (* a b)
  (if (= b 0)
      0
      (+ a (* a (- b 1)))))
```

This algorithm takes a number of steps that is linear in b. Now suppose we include, together with addition, operations double, which doubles an integer, and halve, which divides an (even) integer by 2. Using these, design a multiplication procedure analogous to fast-expt that uses a logarithmic number of steps.

2.29.2 Answer

```scheme
(define (double x)
  (+ x x))
(define (halve x)
  (/ x 2))
(define (fast-mult-rec a b)
  (cond ((= b 0) 0)
        ((even? b)
         (double (fast-mult-rec a (halve b)))) ; This was kind of a stretch to think of.G
         ;(fast-mult (double a) (halve b))) <== My first instinct is iterative
        (else (+ a (fast-mult-rec a (- b 1))))))
```

Proof it works:

```scheme
<<fast-mult-rec>>
<<try-these>>
(try-these (λ(x) (fast-mult-rec 3 x)) (cdr (iota 11)))
```

1	3
2	6
3	9
4	12
5	15
6	18
7	21
8	24
9	27
10	30

2.30 Exercise 1.18

2.30.1 Question

Using the results of Exercise 1.16 and Exercise 1.17, devise a procedure that generates an iterative process for multiplying two integers in terms of adding, doubling, and halving and uses a logarithmic number of steps.

2.30.2 Diary

1. Comparison benchmarks:

```scheme
(load "../mattbench.scm")
<<fast-mult-iter>>
<<fast-mult-rec>>
<<print-table>>
(print-table (list (list "fast-mult-rec" "fast-mult-iter")
                   (list (mattbench (λ() (fast-mult-rec 32
  32)) 10000000)
                         (mattbench (λ() (fast-mult 32 32))
  10000000)))
             #:colnames #t)
```

"fast-mult-rec"	"fast-mult-iter"
196.89	166.35

So the iterative version takes 0.84 times less to do 32×32.

2. Hall of shame Some of my *very* incorrect ideas:

$$ab = (a + 1)(b - 1)$$

$$ab = \left(a + \left(\frac{a}{2}\right)(b - 1)\right)$$

$$ab + c = \left(a(b - 1) + (b + c)\right)$$

2.30.3 Answer

```scheme
(define (double x)
  (+ x x))
(define (halve x)
  (/ x 2))
(define (fast-mult a b)
  (define (iter a b c)
    (cond ((= b 0) 0)
          ((= b 1) (+ a c))
          ((even? b)
           (iter (double a) (halve b) c))
          (else (iter a (- b 1) (+ a c)))))
  (iter a b 0))
```

```scheme
<<fast-mult-iter>>
<<try-these>>
(try-these (λ(x) (fast-mult 3 x)) (cdr (iota 11)))
```

1	3
2	6
3	9
4	12
5	15
6	18
7	21
8	24
9	27
10	30

2.31 Exercise 1.19

2.31.1 Question

There is a clever algorithm for computing the Fibonacci numbers in a logarithmic number of steps. Recall the transformation of the state variables a and b in the `fib-iter` process of section 1-2-2:

$a < -a + b$ and $b < -a$

Call this transformation T, and observe that applying T over and over again n times, starting with 1 and 0, produces the pair _Fib$_{(\mathrm{n}\ +\ 1)}$ and _Fib$_{(\mathrm{n})}$. In other words, the Fibonacci numbers are produced by applying T^n, the nth power of the transformation T, starting with the pair (1,0). Now consider T to be the special case of p = 0 and q = 1 in a family of transformations $T_{(pq)}$, where $T_{(pq)}$ transforms the pair (a,b) according to $a < -bq + aq + ap$ and $b < -bp + aq$. Show that if we apply such a transformation $T_{(pq)}$ twice, the effect is the same as using a single transformation $T_{(p'q')}$ of the same form, and compute p' and q' in terms of p and q. This gives us an explicit way to square these transformations, and thus we can compute T^n using successive squaring, as in the 'fast-expt' procedure. Put this all together to complete the following procedure, which runs in a logarithmic number of steps:

```
(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
                   <??>      ; compute p'
```

```
                    <??>      ; compute q'
                    (/ count 2)))
        (else (fib-iter (+ (* b q) (* a q) (* a p))
                        (+ (* b p) (* a q))
                        p
                        q
                        (- count 1)))))
```

2.31.2 Diary

More succinctly put:

$$\text{Fib}_n \begin{cases} a \leftarrow a + b \\ b \leftarrow a \end{cases}$$

$$\text{Fib-iter}_{abpq} \begin{cases} a \leftarrow bq + aq + ap \\ b \leftarrow bp + aq \end{cases}$$

(T) returns a transformation function based on the two numbers in the attached list. so (T 0 1) returns a fib function.

```
(define (T p q)
  (λ (a b)
    (cons (+ (* b q) (* a q) (* a p))
          (+ (* b p) (* a q)))))

(define T-fib
  (T 0 1))

;; Repeatedly apply T functions:
(define (Tr f n)
  (Tr-iter f n 0 1))
(define (Tr-iter f n a b)
  (if (= n 0)
      a
      (let ((l (f a b)))
        (Tr-iter f (- n 1) (car l) (cdr l)))))
```

$$\text{T}_{pq} : a, b \mapsto \begin{cases} a \leftarrow bq + aq + ap \\ b \leftarrow bp + aq \end{cases}$$

```
<<T-func>>
<<try-these>>
(try-these (λ (x) (Tr (T 0 1) x)) (cdr (iota 11)))
```

1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	34
10	55

2.31.3 Answer

```
(define (fib-rec n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib-rec (- n 1))
                 (fib-rec (- n 2))))))
(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))      ; compute p'
                   (+ (* p q)
                      (* q q)
                      (* q p))      ; compute q'
                   (/ count 2)))
        (else (fib-iter (+ (* b q) (* a q) (* a p))
                        (+ (* b p) (* a q))
                        p
                        q
                        (- count 1)))))
```

"n"	"fib-rec"	"fib-iter"
1	1	1
2	1	1
3	2	2
4	3	3
5	5	5
6	8	8
7	13	13
8	21	21
9	34	34

2.32 1.2.5: Greatest Common Divisor

A greatest common divisor (or GCD) for two integers is the largest integer that divides both of them. A GCD can be quickly found by transforming the problem like so:

$$a\%b = r$$

$$\mathrm{GCD}(a, b) = \mathrm{GCD}(b, r)$$

This eventually produces a pair where the second number is 0. Then, the GCD is the other number in the pair. This is Euclid's Algorithm.

$$\begin{aligned} \mathrm{GCD}(206, 40) &= \mathrm{GCD}(40, 6) \\ &= \mathrm{GCD}(6, 4) \\ &= \mathrm{GCD}(4, 2) \\ &\mathrm{GCD}(2, 0)\ 2 \end{aligned}$$

Lamé's Theorem: If Euclid's Algorithm requires k steps to compute the GCD of some pair, then the smaller number in the pair must be greater than or equal to the k^{th}Fibonacci number.

2.33 Exercise 1.20

2.33.1 Text

```
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b))))
```

2.33.2 Question

The process that a procedure generates is of course dependent on the rules used by the interpreter. As an example, consider the iterative `gcd` procedure given above. Suppose we were to interpret this procedure using normal-order evaluation, as discussed in 1.1.5. (The normal-order-evaluation rule for `if` is described in Exercise 1.5.) Using the substitution method (for normal order), illustrate the process generated in evaluating `(gcd 206 40)` and indicate the remainder operations that are actually performed. How many remainder operations are actually performed in the normal-order evaluation of `(gcd 206 40)`? In the applicative-order evaluation?

2.33.3 Answer

I struggled to understand this, but the key here is that normal-order evaluation causes the unevaluated expressions to be duplicated, meaning they get evaluated multiple times.

1. Applicative order

```
call (gcd 206 40)
(if)
(gcd 40 (remainder 206 40))
eval remainder before call
call (gcd 40 6)
(if)
(gcd 6 (remainder 40 6))
eval remainder before call
call (gcd 6 4)
(if)
(gcd 2 (remainder 4 2))
eval remainder before call
call (gcd 2 0)
(if)
;; => 2
```

```
;; call gcd
(gcd 206 40)

;; eval conditional
(if (= 40 0)
    206
    (gcd 40 (remainder 206 40)))

```

```scheme
;; recurse
(gcd 40 (remainder 206 40))

; encounter conditional
(if (= (remainder 206 40) 0)
    40
    (gcd (remainder 206 40)
         (remainder 40 (remainder 206 40))))

; evaluate 1 remainder
(if (= 6 0)
    40
    (gcd (remainder 206 40)
         (remainder 40 (remainder 206 40))))

; recurse
(gcd (remainder 206 40)
     (remainder 40 (remainder 206 40)))

; encounter conditional
(if (= (remainder 40 (remainder 206 40)) 0)
    (remainder 206 40)
    (gcd (remainder 40 (remainder 206 40))
         (remainder (remainder 206 40) (remainder 40 (remainder 206 40)))))

; eval 2 remainder
(if (= 4 0)
    (remainder 206 40)
    (gcd (remainder 40 (remainder 206 40))
         (remainder (remainder 206 40) (remainder 40 (remainder 206 40)))))

; recurse
(gcd (remainder 40 (remainder 206 40))
     (remainder (remainder 206 40) (remainder 40 (remainder 206 40))))

; encounter conditional
(if (= (remainder (remainder 206 40) (remainder 40 (remainder 206 40))) 0)
    (remainder 40 (remainder 206 40))
    (gcd (remainder (remainder 206 40) (remainder 40 (remainder 206 40)))
```

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

; eval 4 remainders
(if (= 2 0)
    (remainder 40 (remainder 206 40))
    (gcd (remainder (remainder 206 40) (remainder 40 (remainder 206 40)))
         (remainder (remainder 40 (remainder 206 40)) (remainder (remainder 206 40) (remainder 40 (remainder 206 40))))))

; recurse
(gcd (remainder (remainder 206 40) (remainder 40 (remainder 206 40)))
     (remainder (remainder 40 (remainder 206 40)) (remainder (remainder 206 40) (remainder 40 (remainder 206 40)))))

; encounter conditional
(if (= (remainder (remainder 40 (remainder 206 40)) (remainder (remainder 206 40) (remainder 40 (remainder 206 40)))) 0)
    (remainder (remainder 206 40) (remainder 40 (remainder 206 40)))
    (gcd (remainder (remainder 40 (remainder 206 40)) (remainder (remainder 206 40) (remainder 40 (remainder 206 40)))) (remainder a  (remainder (remainder 40 (remainder 206 40)) (remainder (remainder 206 40) (remainder 40 (remainder 206 40)))))))

; eval 7 remainders
(if (= 0 0)
    (remainder (remainder 206 40) (remainder 40 (remainder 206 40)))
    (gcd (remainder (remainder 40 (remainder 206 40)) (remainder (remainder 206 40) (remainder 40 (remainder 206 40)))) (remainder a  (remainder (remainder 40 (remainder 206 40)) (remainder (remainder 206 40) (remainder 40 (remainder 206 40)))))))

; eval 4 remainders
```

```
(remainder (remainder 206 40) (remainder 40 (remainder 206
↪  40)))
; => 2
```

So, in normal-order eval, remainder is called 18 times, while in applicative order it's called 5 times.

2.34 1.2.6: Example: Testing for Primality

Two algorithms for testing primality of numbers.

1. $\Theta(\sqrt{n})$: Start with $x = 2$, check for divisibility with n, if not then increment x by 1 and check again. If $x^2 > n$ and you haven't found a divisor, n is prime.

2. $\Theta(\log n)$: 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.

   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. [Two numbers are *congruent modulo* n if they both have the same remainder when divided by n.]

The Fermat test is a probabilistic algorithm, meaning its answer is likely to be correct rather than guaranteed to be correct. Repeating the test increases the likelihood of a correct answer.

2.35 Exercise 1.21

2.35.1 Text

```
<<square>>
(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))
```

2.35.2 Question

Use the smallest-divisor procedure to find the smallest divisor of each of the following numbers: 199, 1999, 19999.

2.35.3 Answer

```
<<find-divisor-txt>>
(map smallest-divisor '(199 1999 19999))
```

199	1999	7

2.36 Exercise 1.22

2.36.1 Question

Most Lisp implementations include a primitive called runtime that returns an integer that specifies the amount of time the system has been running (measured, for example, in microseconds). The following timed-prime-test procedure, when called with an integer n, prints n and checks to see if n is prime. If n is prime, the procedure prints three asterisks followed by the amount of time used in performing the test.

```
<<find-divisor-txt>>
(define (prime? n)
  (= n (smallest-divisor n)))
```

```
<<prime-smallest-divisor>>
(define (timed-prime-test n)
  (newline)
  (display n) ;; Guile compatible \downarrow
  (start-prime-test n (get-internal-run-time)))
(define (start-prime-test n start-time)
  (if (prime? n)
      (begin
        (report-prime (- (get-internal-run-time)
                         start-time))
        n)
```

```
      #f))
(define (report-prime elapsed-time)
  (display " *** ")
  (display elapsed-time))
```

Using this procedure, write a procedure search-for-primes that checks the primality of consecutive odd integers in a specified range. Use your procedure to find the three smallest primes larger than 1000; larger than 10,000; larger than 100,000; larger than 1,000,000. Note the time needed to test each prime. Since the testing algorithm has order of growth of $\Theta(\sqrt{n})$, you should expect that testing for primes around 10,000 should take about $\sqrt{10}$ times as long as testing for primes around 1000. Do your timing data bear this out? How well do the data for 100,000 and 1,000,000 support the $\Theta(\sqrt{n})$ prediction? Is your result compatible with the notion that programs on your machine run in time proportional to the number of steps required for the computation?

2.36.2 Answer

1. Part 1 So this question is a little funky, because modern machines are so fast that the single-run times can seriously vary.

```
<<timed-prime-test-txt>>
(define (search-for-primes minimum goal)
  (define m (if (even? minimum)
                (+ minimum 1)
                (minimum)))
  (search-for-primes-iter m '() goal))
(define (search-for-primes-iter n lst goal)
  (if (= goal 0)
      lst
      (let ((x (timed-prime-test n)))
        (if (not (equal? x #f))
            (search-for-primes-iter (+ n 2) (cons x lst) (-
 ↪  goal 1))
            (search-for-primes-iter (+ n 2) lst goal)))))
```

```
<<search-primes-basic>>
(let ((lt1000-1 (search-for-primes 1000 3)))
  (list "Primes > 1000" lt1000-1))
```

```
1001
1003
1005
1007
1009 *** 1651
1011
1013 *** 1425
1015
1017
1019 *** 1375
```

There's proof it works. And here are the answers to the question:

```
<<search-primes-basic>>
(let ((lt1000-1 (search-for-primes 1000 3))
      (lt10000-1 (search-for-primes 10000 3))
      (lt100000-1 (search-for-primes 100000 3))
      (lt100000000-1 (search-for-primes 1000000 3)))
  (list
   (list "Primes > 1000" (reverse lt1000-1))
   (list "Primes > 10000" (reverse lt10000-1))
   (list "Primes > 100000" (reverse lt100000-1))
   (list "Primes > 100000000" (reverse lt100000000-1))
   ))
```

Primes > 1000	(1009 1013 1019)
Primes > 10000	(10007 10009 10037)
Primes > 100000	(100003 100019 100043)
Primes > 100000000	(1000003 1000033 1000037)

2. Part 2 Repeatedly re-running, it I see it occasionally jump to twice the time. I'm not happy with this, so I'm going to refactor to use the `mattbench2` utility from the root of the project folder.

```
(define (mattbench2 f n)
  ;; Executes "f" for n times, and returns how long it took.
  ;; f is a lambda that takes no arguments, a.k.a. a "thunk"

  ;; Returns a list with car(last execution results) and
  ↪  cadr(time taken divided by iterations n)

  (define (time-getter) (get-internal-run-time))
  (define start-time (time-getter))
  (define (how-long) (- (time-getter) start-time))

```

```
  (define (iter i)
    (f)
    (if (<= i 0)
        (f) ;; return the results of the last function call
        (iter (- i 1))))

  (list (iter n) ;; result of last call of f
        (/ (how-long) (* n 1.0))));; Divide by iterations so changed n has no effect
```

I'm going to get some more precise times. First, I need a prime searching variant that doesn't bother benchmarking. This will call `prime?`, which will be bound later since we'll be trying different methods.

```
(define (search-for-primes minimum goal)
  (define m (if (even? minimum)
                (+ minimum 1)
                (minimum)))
  (search-for-primes-iter m '() goal))
(define (search-for-primes-iter n lst goal)
  (if (= goal 0)
      lst
      (let ((x (prime? n)))
        (if (not (equal? x #f))
            (search-for-primes-iter (+ n 2) (cons n lst) (- goal 1))
            (search-for-primes-iter (+ n 2) lst goal)))))
```

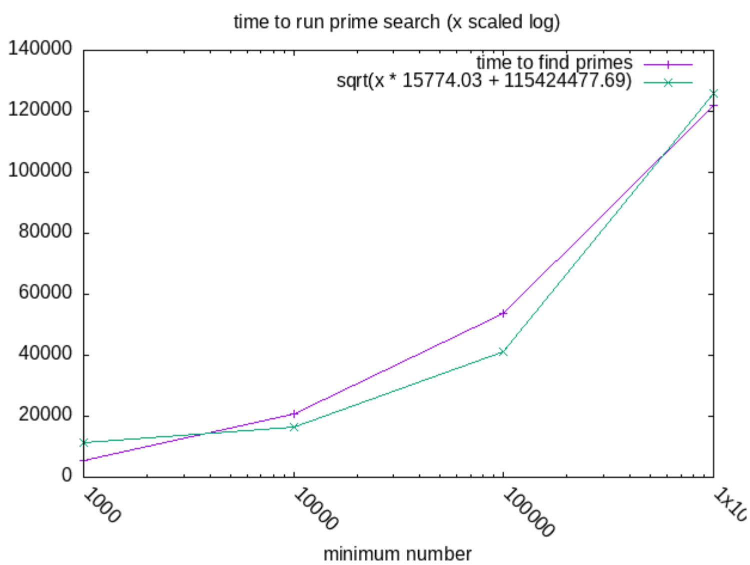
I can benchmark these functions like so:

```
<<mattbench2>>
<<prime-smallest-divisor>>
<<search-for-primes-untimed>>
<<print-table>>

(define benchmark-iterations 1000000)
(define (testit f)
  (list (cadr (mattbench2 (λ() (f 1000 3)) benchmark-iterations))
        (cadr (mattbench2 (λ() (f 10000 3)) benchmark-iterations))
        (cadr (mattbench2 (λ() (f 100000 3)) benchmark-iterations))
        (cadr (mattbench2 (λ() (f 1000000 3)) benchmark-iterations))))
```

```
(print-row
 (testit search-for-primes))
```

Here are the results (run externally from Org-Mode):

5425.223086	20772.332491	53577.240193	121986.712395



The plot for the square root function doesn't quite fit the real one and I'm not sure where the fault lies. I don't struggle to understand things like "this algorithm is slower than this other one," but when asked to find or prove the Θ notation I'm pretty clueless;

2.37 Exercise 1.23

2.37.1 Question

The `smallest-divisor` procedure shown at the start of this section does lots of needless testing: After it checks to see if the number is divisible by 2 there is no point in checking to see if it is divisible by any larger even numbers. This suggests that the values used for test-divisor should not be 2, 3, 4, 5, 6, …, but rather 2, 3, 5, 7, 9, …. To implement this change, define a procedure `next` that returns 3 if its input is equal to 2 and otherwise returns its input plus 2. Modify the smallest-divisor procedure to use `(next test-divisor)` instead of `(+ test-divisor 1)`. With `timed-prime-test` incorporating this modified version of `smallest-divisor`, run the test for each of the 12 primes found in Exercise 1.22. Since this modification halves the number of test steps, you should expect it to run about twice as fast. Is this expectation confirmed? If not, what is the observed ratio of the speeds of the two algorithms, and how do you explain the fact that it is different from 2?

2.37.2 A Comedy of Error (just the one)

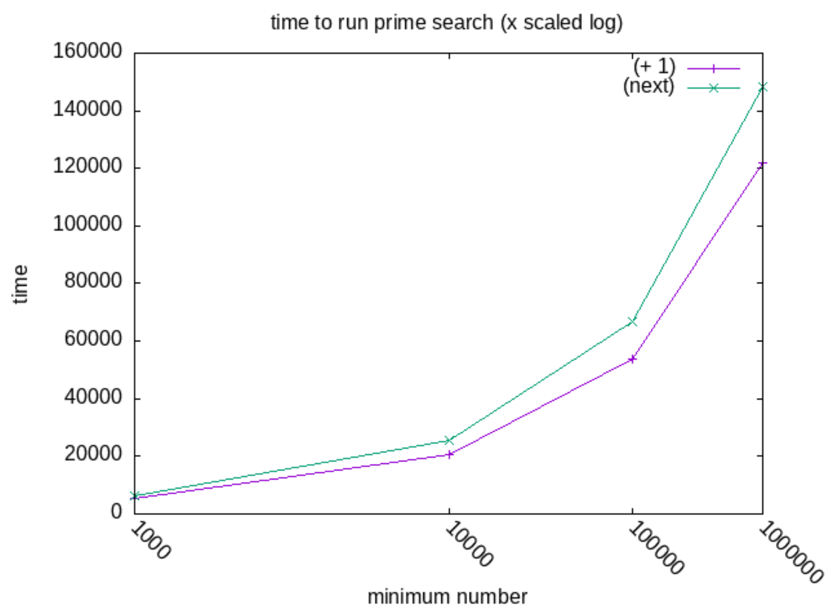
```
<<square>>
(define (smallest-divisor n)
  (find-divisor n 2))

(define (next n)
  (if (= n 2)
      3
      (+ n 1)))

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

```
<<mattbench2>>
<<find-divisor-faster>>
(define (prime? n)
  (= n (smallest-divisor n)))
<<search-for-primes-untimed>>
<<print-table>>

(define benchmark-iterations 1000000)
(define (testit f)
  (list (cadr (mattbench2 (λ() (f 1000 3)) benchmark-iterations))
        (cadr (mattbench2 (λ() (f 10000 3)) benchmark-iterations))
        (cadr (mattbench2 (λ() (f 100000 3)) benchmark-iterations))
```

```
        (cadr (mattbench2 (λ() (f 1000000 3))
  benchmark-iterations))))

(print-row
 (testit search-for-primes))
```

6456.538118	25550.757304	66746.041644	148505.580638

min	(+1)	(next)
1000	5507.42497	6366.99462
10000	20913.71497	24845.9193
100000	53778.74737	64756.73693
1000000	122135.60511	143869.63561



So it's *slower* than before. Why?
Oh, that's why.

```
(define (next n)
  (if (= n 2)
      3
      (+ n 1))) ;; <-- D'oh.
```

2.37.3 Answer

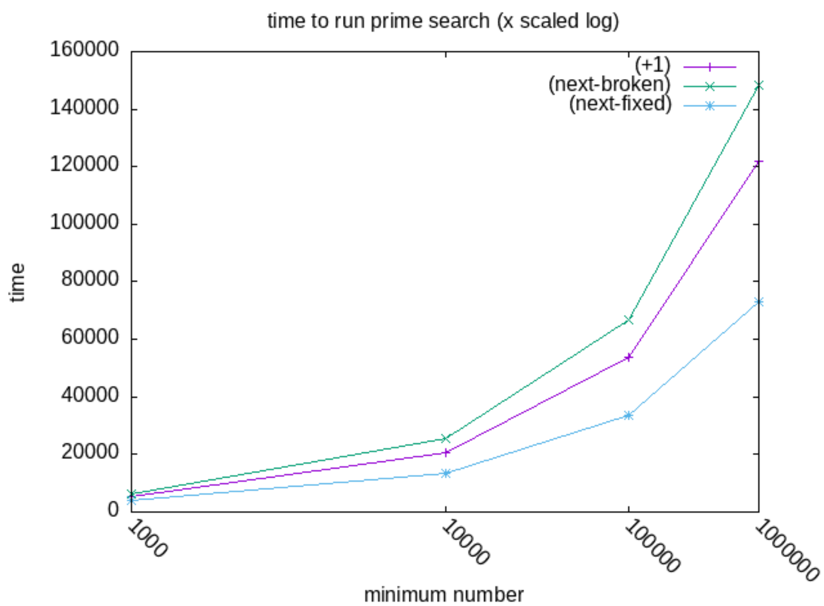
Ok, let's try that again.

```scheme
<<square>>
(define (smallest-divisor n)
  (find-divisor n 2))

(define (next n)
  (if (= n 2)
      3
      (+ n 2)))

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

```scheme
<<mattbench2>>
<<find-divisor-faster-real>>
(define (prime? n)
  (= n (smallest-divisor n)))
<<search-for-primes-untimed>>
<<print-table>>

(define benchmark-iterations 500000)
(define (testit f)
  (list (cadr (mattbench2 (λ() (f 1000 3)) benchmark-iterations))
        (cadr (mattbench2 (λ() (f 10000 3)) benchmark-iterations))
        (cadr (mattbench2 (λ() (f 100000 3)) benchmark-iterations))
        (cadr (mattbench2 (λ() (f 1000000 3)) benchmark-iterations))))

(print-row
 (testit search-for-primes))
```

3863.7424	13519.209814	33520.676384	73005.539932

min	(+1)	(next-broken)	(next-fixed)
—	—	—	—
1000	5425.223086	6456.538118	3863.7424
10000	20772.332491	25550.757304	13519.209814
100000	53577.240193	66746.041644	33520.676384
1000000	121986.712395	148505.580638	73005.539932



I had a lot of trouble getting this one to compile, I have to restart Emacs in order to get it to render.

Anyways, there's the speedup that was expected. Let's compare the ratios.

Defining a new average that takes arbitrary numbers of arguments:

```
(define (average . args)
  (let ((len (length args)))
    (/ (apply + args) len)))
```

Using it for percentage comparisons:

```
<<average-varargs>>
(list (cons "% speedup for broken (next)"
            (cons (format #f "~2$%"
                          (apply average
                                 (map (λ (x y) (* 100 (/ x y)))
```

```
                                      (car smd) (car smdf))))
                  #nil))
      (cons "% speedup for real (next)"
            (cons (format #f "~2$%"
                          (apply average
                                 (map (λ (x y) (* 100 (/ x y)))
                                      (car smd) (car smdff))))
                  #nil)))
```

% speedup for broken (next)	81.93%
% speedup for real (next)	155.25%

Since this changed algorithm cuts out almost half of the steps, you might expect something more like a 200% speedup. Let's try optimizing it further. Two observations:

1. The condition `(divides? 2 n)` only needs to be run once at the start of the program.
2. Because it only needs to be run once, it doesn't need to be a separate function at all.

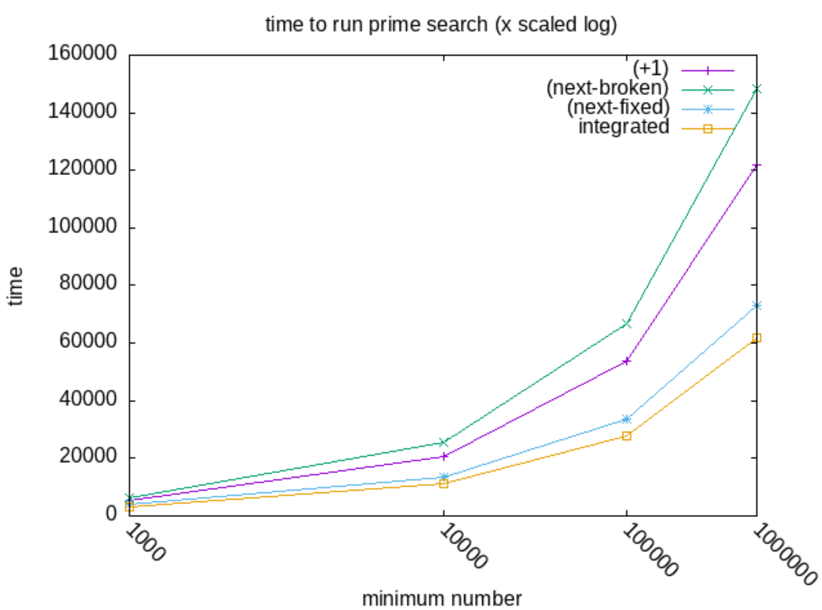
```
<<square>>
(define (smallest-divisor n)
  (if (divides? 2 n)                  ;; check for division by 2
      2
      (find-divisor n 3)))            ;; start find-divisor at 3

(define (find-divisor n test-divisor)
  (cond ((> (square test-divisor) n)
         n)
        ((divides? test-divisor n)
         test-divisor)
        (else (find-divisor
               n
               (+ 2 test-divisor))))) ;; just increase by 2

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

```
<<mattbench2>>
<<find-divisor-faster-real2>>
(define (prime? n)
  (= n (smallest-divisor n)))
<<search-for-primes-untimed>>
```

```
<<print-table>>

(define benchmark-iterations 500000)
(define (testit f)
  (list (cadr (mattbench2 (λ() (f 1000 3)) benchmark-iterations))
        (cadr (mattbench2 (λ() (f 10000 3)) benchmark-iterations))
        (cadr (mattbench2 (λ() (f 100000 3)) benchmark-iterations))
        (cadr (mattbench2 (λ() (f 1000000 3)) benchmark-iterations))))

(print-row
 (testit search-for-primes))
```

3151.259574	11245.20428	27803.067944	61997.275154

min	(+1)	(next-broken)	(next-fixed)	integrated
—	—	—	—	—
1000	5425.223086	6456.538118	3863.7424	3151.259574
10000	20772.332491	25550.757304	13519.209814	11245.20428
100000	53577.240193	66746.041644	33520.676384	27803.067944
1000000	121986.712395	148505.580638	73005.539932	61997.275154



% speedup for broken (next)	81.93%
% speedup for real (next)	155.25%
% speedup for optimized	186.59%

2.38 Exercise 1.24

2.38.1 Text

```
<<square>>
(define (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
         (remainder
          (square (expmod base (/ exp 2) m))
          m))
        (else
         (remainder
          (* base (expmod base (- exp 1) m))
          m))))
```

```
(define (fermat-test n)
  (define (try-it a)
    (= (expmod a n n) a))
  (try-it (+ 1 (random (- n 1)))))
```

```
(define (fast-prime? n times)
  (cond ((= times 0) #t)
        ((fermat-test n)
         (fast-prime? n (- times 1)))
        (else #f)))
```

2.38.2 Question

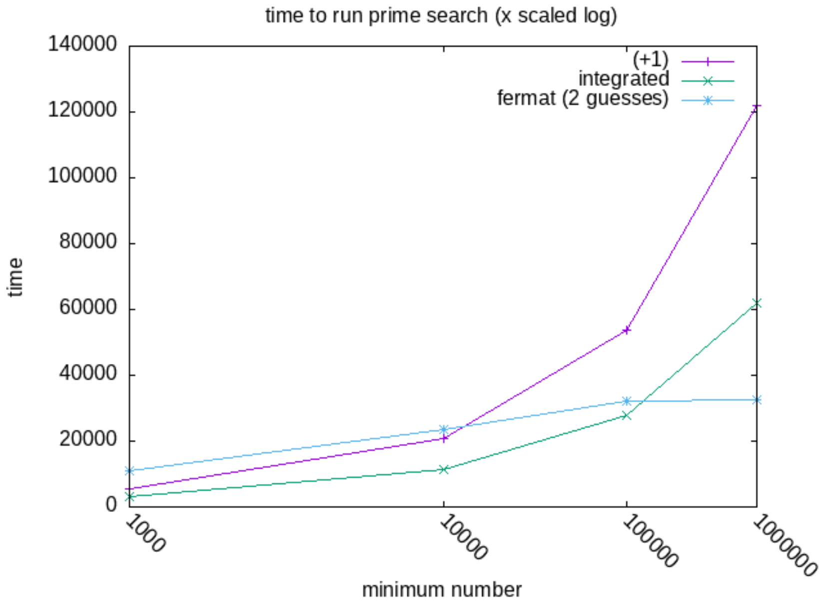
Modify the `timed-prime-test` procedure of Exercise 1.22 to use `fast-prime?` (the Fermat method), and test each of the 12 primes you found in that exercise. Since the Fermat test has $\Theta(\log n)$ growth, how would you expect the time to test primes near 1,000,000 to compare with the time needed to test primes near 1000? Do your data bear this out? Can you explain any discrepancy you find?

2.38.3 Answer

```
<<mattbench2>>
<<expmod>>
<<fermat-test>>
<<fast-prime>>
(define fermat-iterations 2)
(define (prime? n)
  (fast-prime? n fermat-iterations))
<<search-for-primes-untimed>>
<<print-table>>

(define benchmark-iterations 500000)
(define (testit f)
  (list (cadr (mattbench2 (λ() (f 1000 3)) benchmark-iterations))
        (cadr (mattbench2 (λ() (f 10000 3)) benchmark-iterations))
        (cadr (mattbench2 (λ() (f 100000 3)) benchmark-iterations))
        (cadr (mattbench2 (λ() (f 1000000 3)) benchmark-iterations))))

(print-row
 (testit search-for-primes))
```

11175.799722	23518.62116	32150.745642	32679.766448

min	(+1)	integrated	fermat (2 guesses)
—	—	—	—
1000	5425.223086	3151.259574	11175.799722
10000	20772.332491	11245.20428	23518.62116
100000	53577.240193	27803.067944	32150.745642
1000000	121986.712395	61997.275154	32679.766448



It definitely looks to be advancing much slower than the other methods. I'd like to see more of the function.

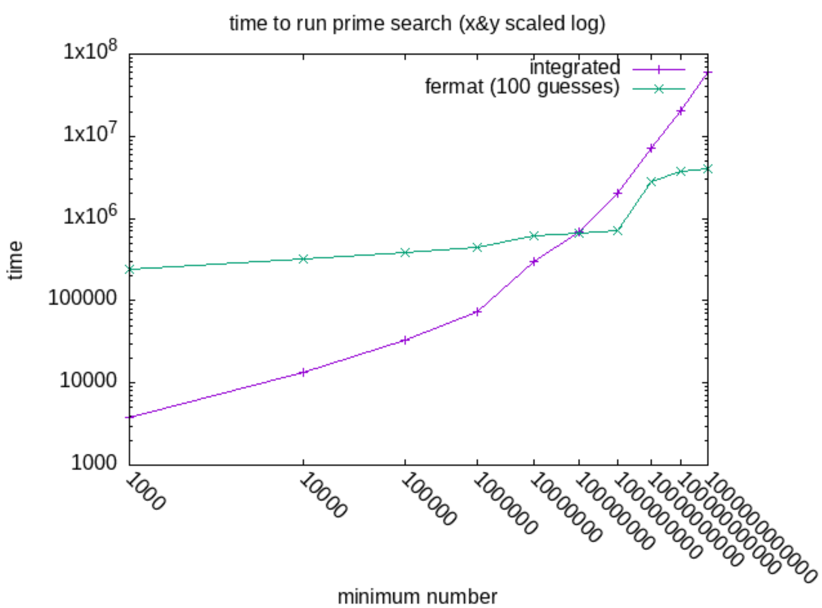
```
<<mattbench2>>
<<find-divisor-faster-real>>
(define (prime? n)
  (= n (smallest-divisor n)))
<<search-for-primes-untimed>>
<<print-table>>

(define benchmark-iterations 100000)
(define (testit f)
  (list (cadr (mattbench2 (λ() (f 1000 3)) benchmark-iterations))
        (cadr (mattbench2 (λ() (f 10000 3)) benchmark-iterations))
        (cadr (mattbench2 (λ() (f 100000 3)) benchmark-iterations))
        (cadr (mattbench2 (λ() (f 1000000 3)) benchmark-iterations))
        (cadr (mattbench2 (λ() (f 10000000 3)) benchmark-iterations))
        (cadr (mattbench2 (λ() (f 100000000 3)) benchmark-iterations))
        (cadr (mattbench2 (λ() (f 1000000000 3)) benchmark-iterations))
```

```
        (cadr (mattbench2 (λ() (f 10000000000 3)) benchmark-iterations))
        (cadr (mattbench2 (λ() (f 100000000000 3)) benchmark-iterations))
        (cadr (mattbench2 (λ() (f 1000000000000 3)) benchmark-iterations))))

(print-row
 (testit search-for-primes))
```

```
<<mattbench2>>
<<expmod>>
<<fermat-test>>
<<fast-prime>>
(define fermat-iterations 100)
(define (prime? n)
  (fast-prime? n fermat-iterations))
<<search-for-primes-untimed>>
<<print-table>>

(define benchmark-iterations 100000)
(define (testit f)
  (list (cadr (mattbench2 (λ() (f 1000 3)) benchmark-iterations))
        (cadr (mattbench2 (λ() (f 10000 3)) benchmark-iterations))
        (cadr (mattbench2 (λ() (f 100000 3)) benchmark-iterations))
        (cadr (mattbench2 (λ() (f 1000000 3)) benchmark-iterations))
        (cadr (mattbench2 (λ() (f 10000000 3)) benchmark-iterations))
        (cadr (mattbench2 (λ() (f 100000000 3)) benchmark-iterations))
        (cadr (mattbench2 (λ() (f 1000000000 3)) benchmark-iterations))
        (cadr (mattbench2 (λ() (f 10000000000 3)) benchmark-iterations))
        (cadr (mattbench2 (λ() (f 100000000000 3)) benchmark-iterations))
        (cadr (mattbench2 (λ() (f 1000000000000 3)) benchmark-iterations))))

(print-row
 (testit search-for-primes))
```

3802.45146	13397.91871	32948.31241	73237.64777	299326.76182	678512.75719	2064911.33345
237945.8945	319761.90842	391573.47557	448501.96232	614009.08547	661205.34772	700058.30723

min	integrated	fermat (100 guesses)
—	—	—
1000	3802.45146	237945.8945
10000	13397.91871	319761.90842
100000	32948.31241	391573.47557
1000000	73237.64777	448501.96232
10000000	299326.76182	614009.08547
100000000	678512.75719	661205.34772
1000000000	2064911.33345	700058.30723
10000000000	7065717.58395	2852221.29076
100000000000	20198370.27007	3717690.96246
1000000000000	60956807.83034	3995948.05596



For the life of me I have no idea what that bump is. Maybe it needs more aggressive bignum processing there?

2.39 Exercise 1.25

2.39.1 Question

> Alyssa P. Hacker complains that we went to a lot of extra work in writing `expmod`. After all, she says, since we already know how to compute exponentials, we could have simply written

```scheme
(define (expmod base exp m)
  (remainder (fast-expt base exp) m))
```

Is she correct? Would this procedure serve as well for our fast prime tester? Explain.

2.39.2 Answer

In Alyssa's version of `expmod`, the result of the `fast-expt` operation is *extremely* large. For example, in the process of checking for divisors of 1,001, the number 455 will be tried. `(expt 455 1001)` produces an integer 2,661 digits long. This is just one of the thousands of exponentiations that `smallest-divisor` will perform. It's best to avoid this, so we use to our advantage the fact that we only need to know the remainder of the exponentiations. `expmod` breaks down the exponentiation into smaller steps and performs `remainder` after every step, significantly reducing the memory requirements.

As an example, let's trace (some of) the execution of `(expmod 455 1001 1001)`:

```scheme
(expmod 455 1001 1001)
>  (even? 1001)
>  #f
>  (expmod 455 1000 1001)
>  >  (even? 1000)
>  >  #t
>  >  (expmod 455 500 1001)
>  >  >  (even? 500)
>  >  >  #t
;; ...
>  >  >  x11 (expmod 455 2 1001)
>  >  >  x11 >  (even? 2)
>  >  >  x11 >  #t
>  >  >  x11 >  (expmod 455 1 1001)
>  >  >  x11 >  >  (even? 1)
>  >  >  x11 >  >  #f
>  >  >  x11 >  >  (expmod 455 0 1001)
>  >  >  x11 >  >  1
>  >  >  x11 >  455
>  >  >  x11 >  (square 455)
>  >  >  x11 >  207025
>  >  >  x11 819
;; ...
>  >  >  (square 364)
>  >  >  132496
>  >  364
```

```
> > (square 364)
> > 132496
> 364
455
```

You can see that the numbers remain quite manageable throughout this process. So taking these extra steps actually leads to an algorithm that performs better.

2.40 Exercise 1.26

2.40.1 Question

> Louis Reasoner is having great difficulty doing Exercise 1.24. His `fast-prime?` test seems to run more slowly than his `prime?` test. Louis calls his friend Eva Lu Ator over to help. When they examine Louis's code, they find that he has rewritten the `expmod` procedure to use an explicit multiplication, rather than calling `square`:

```
(define (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
         (remainder
          (* (expmod base (/ exp 2) m) ;; <== hmm.
             (expmod base (/ exp 2) m))
          m))
        (else
         (remainder
          (* base
             (expmod base (- exp 1) m))
          m))))
```

> "I don't see what difference that could make," says Louis. "I do." says Eva. "By writing the procedure like that, you have transformed the $\Theta(\log n)$ process into a $\Theta(n)$ process." Explain.

2.40.2 Answer

Making the same function call twice isn't the same as using a variable twice – Louis' version doubles the work, having two processes solving the exact same problem. Since the number of processes used increases exponentially, this turns $\log n$ into n.

2.41 Exercise 1.27

2.41.1 Question

Demonstrate that the Carmichael numbers listed in Footnote 1.47 really do fool the Fermat test. That is, write a procedure that takes an integer n and tests whether a^n is congruent to a modulo n for every $a < n$, and try your procedure on the given Carmichael numbers.

$$561 \quad 1105 \quad 1729 \quad 2465 \quad 2821 \quad 6601$$

2.41.2 Answer

```
<<expmod>>
(define (car-test n)
  (define (check a)
    (= (remainder (expt a n) n)
       (remainder (modulo a n) n)))
  (every check
         (cddr (iota n))))
```

```
<<car-test>>
(list (car-test 12) ; <== false (not prime)
      (car-test 1009);<== true  (real prime)
      (car-test 561));<== true  (not prime,
                     ;      Carmichael number)
```

2.42 Exercise 1.28

2.42.1 Question

One variant of the Fermat test that cannot be fooled is called the Miller-Rabin test (Miller 1976; Rabin 1980). This starts from an alternate form of Fermat's Little Theorem, which states that if n is a prime number and a is any positive integer less than n, then a raised to the $(n-1)$ -st power is congruent to 1 modulo n. To test the primality of a number n by the Miller-Rabin test, we pick a random number $a < n$ and raise a to the $(n-1)$ -st power modulo n using the `expmod` procedure. However, whenever we perform the squaring step in `expmod`, we check to see if we have discovered a "nontrivial square root of 1 modulo n," that is, a number not equal to 1 or $n-1$ whose square is equal to 1 modulo n. It is possible to prove that if such a nontrivial square root of 1 exists, then n is not prime. It is also possible to prove that if n is an odd number that is not prime, then, for at least half the numbers $a < n$, computing $an-1$ in this way will reveal a nontrivial square root of 1 modulo n. (This is why the Miller-Rabin test cannot be fooled.) Modify the `expmod` procedure to signal if it discovers a nontrivial square root of 1, and use this to implement the Miller-Rabin test with a procedure analogous to fermat-test. Check your procedure by testing various known primes and non-primes. Hint: One convenient way to make `expmod` signal is to have it return 0.

2.42.2 Analysis

For the sake of verifying this, I want to get a bigger list of primes and Carmichael numbers to verify against. I'll save them using Guile's built in read/write functions that save Lisp lists to text:

```
<<find-divisor-faster-real>>
(define (prime? n)
  (= n (smallest-divisor n)))
(call-with-output-file "Data/primes-1k_to_1mil.txt" (λ(port)
  (write (filter prime? (iota (- 1000000 1000) 1000))
         port)))
```

```
;; fermat prime test but checks *every* value from 2 to n-1
(define (fermat-prime? n)
  (define (iter a)
    (if (= a n)
        #f
        (if (= (expmod a n n) a)
            #t
            (iter (+ 1 a)))))
  (iter 2))
```

```
(use-srfis '(1))
<<expmod>>
<<fermat-prime?>>
<<find-divisor-faster-real>>
(define (prime? n)
  (= n (smallest-divisor n)))
(call-with-output-file "Data/carmichael-verification.txt" (λ(port)
     (write (filter
             (λ(x) (and (fermat-prime? x)
                        (not (prime? x))))
```

```
              (iota (- 1000000 1000) 1000))
             port)))
```

This will be useful in various future functions:

```
(define list-of-primes (call-with-input-file
↪  "Data/primes-1k_to_1mil.txt" read))
(define list-of-carmichaels (call-with-input-file
↪  "Data/carmichael.txt" read))
```

```
(use-srfis '(1))
<<expmod>>
<<fermat-prime?>>
<<find-divisor-faster-real>>
(define (prime? n)
  (= n (smallest-divisor n)))
<<get-lists-of-primes>>
(define prime-is-working
  (and (and-map prime? list-of-primes)
       (not (and-map prime? list-of-carmichaels))))
(format #t "(prime?) is working: ~a~%"
        (if prime-is-working
            "Yes"
            "No"))
(define fermat-is-vulnerable
  (and (and-map fermat-prime? list-of-primes)
       (and-map fermat-prime? list-of-carmichaels)))
(format #t "(fermat-prime?) is vulnerable: ~a~%"
        (if fermat-is-vulnerable
            "Yes"
            "No"))
```

```
(prime?) is working: Yes
(fermat-prime?) is vulnerable: Yes
```

2.42.3 Answer

```
<<square>>
(define (expmod-mr base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
         (let ((sqr
                (square (expmod-mr base (/ exp 2) m))))
           (if (= 1 (modulo sqr m))
               0
```

```
              (remainder sqr m))))
        (else
         (remainder
          (* base (expmod-mr base (- exp 1) m))
          m))))
```

```
(define (mr-test n)
  (define (try-it a)
    (let ((it (expmod-mr a n n)))
      (or (= it a)
          (= it 0))))
  (try-it (+ 1 (random (- n 1)))))
```

```
(define (mr-prime? n times)
  (cond ((= times 0) #t)
        ((mr-test n)
         (mr-prime? n (- times 1)))
        (else #f)))
```

```
<<expmod-mr>>
<<mr-test>>
<<mr-prime>>
(define mr-times 100)
<<get-lists-of-primes>>
(format #t "      mr detects primes: ~a~%mr false-positives Carmichaels: ~a~%"
        (and-map (λ(x)(mr-prime? x mr-times)) list-of-primes)
      (and-map (λ(x)(mr-prime? x mr-times)) list-of-carmichaels))
```

```
      mr detects primes: #t
mr false-positives Carmichaels: #t
```

Shoot. And I thought I did a very literal interpretation of what the book asked.

Ah, I see the problem. I need to keep track of what the pre-squaring number was and use that to determine whether the square is valid or not.

```
<<square>>
(define (expmod-mr base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
         ;; Keep result and remainder seperate
         (let* ((result (expmod-mr base (/ exp 2) m))
```

```
              (rem (remainder (square result) m)))
          (if (and (not (= result 1))
                   (not (= result (- m 1)))
                   (= 1 rem))
              0 ;; non-trivial sqrt mod 1 is found
              rem)))
        (else
         (remainder
          (* base (expmod-mr base (- exp 1) m))
          m))))
```

Unfortunately this one has the same problem. What's the issue?

Sadly, there's a massive issue in `mr-test`.

```
(define (mr-test n)
  (define (try-it a)
    (let ((it (expmod-mr a n n))) ;; Should be "a (- n 1) n"
      (or (= it a)    ;; Should be (= it 1)
          (= it 0)))) ;; Two strikes, you're out
  (try-it (+ 1 (random (- n 1)))))
```

One more time.

```
(define (mr-test n)
  (define (try-it a)
    (= 1 (expmod-mr a (- n 1) n)))
  (try-it (+ 1 (random (- n 1)))))
```

```
<<expmod-mr2>>
<<mr-test2>>
<<mr-prime>>
(define mr-times 100)
<<get-lists-of-primes>>
(format #t "      mr detects primes: ~a~%mr false-positives Carmichaels: ~a~%"
        (and-map (λ(x)(mr-prime? x mr-times)) list-of-primes)
      (and-map (λ(x)(mr-prime? x mr-times)) list-of-carmichaels))
```

```
      mr detects primes: #t
mr false-positives Carmichaels: #f
```

2.43 1.3: Formulating Abstractions with Higher-Order Procedures

Procedures that manipulate procedures are called *higher-order procedures*.

2.44 1.3.1: Procedures as Arguments

Let's say we have several different types of series that we want to sum. Functions for each of these tasks will look very similar, so we're better off defining a general function that expresses the *idea* of summation, that can then be passed specific functions to cause the specific behavior of the series. Mathematicians express this as $\sum$ ("sigma") notation.

For the program:

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

Which is equivalent to:

$$\sum_{n\ a}^{b} term(n)\ term(a)+term(next(a))+term(next(next(a)))+\cdots+term(b)$$

We can pass integers to `a` and `b` and functions to `term` and `next`. Note that in order to simply sum integers, we'd need to define and pass an identity function to `term`.

2.45 Exercise 1.29

2.45.1 Text

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

```
(define (integral f a b dx)
  (define (add-dx x)
    (+ x dx))
  (* (sum f (+ a (/ dx 2.0)) add-dx b)
     dx))
```

2.45.2 Question

> Simpson's Rule is a more accurate method of numerical integration than the method illustrated above. Using Simpson's Rule, the integral of a function f between a and b is approximated as

$$\frac{h}{3}(y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \cdots + 2y_{n-2} + 4y_{n-1} + y_n)$$

where $h = (b-a)/n$, for some even integer n, and $y_k = f(a+kh)$. (Increasing n increases the accuracy of the approximation.) Define a procedure that takes as arguments f, a, b, and n and returns the value of the integral, computed using Simpson's Rule. Use your procedure to integrate `cube` between 0 and 1 (with $n = 100$ and $n = 1000$), and compare the results to those of the `integral` procedure shown above.

2.45.3 Answer

```
(define (int-simp f a b n)
  (define h
    (/ (- b a)
     n))
  (define (gety k)
    (f (+ a (* k h))))
  (define (series-y sum k) ;; start with sum = y_0
    (cond ((= k n) (+ sum (gety k)));; and k = 1
          ((even? k) (series-y
                      (+ sum (* 2 (gety k)))
                      (+ 1 k)))
          (else (series-y
                 (+ sum (* 4 (gety k)))
                 (+ 1 k)))))
  (define sum-of-series (series-y (gety a) 1)) ;; (f a) = y_0
  (* (/ h 3) sum-of-series))
```

Let's compare these at equal levels of computational difficulty.

```
<<mattbench2>>
<<print-table>>
(define (cube x)
  (* x x x))
<<sum>>
<<integral>>
<<int-simp>>

(define iterations 100000) ;; benchmark iterations
(define (run-test1)
  (integral cube 0.0 1.0 0.0008))
(define (run-test2)
```

```
  (int-simp cube 0.0 1.0 1000.0))
(print-table (list (list "integral dx:0.0008" "int-simp i:1000")
                   (list (run-test1) (run-test2))
                   (list (cadr (mattbench2 run-test1 iterations))
                         (cadr (mattbench2 run-test2 iterations))))
             #:colnames #t)
```

integral dx:0.0008	int-simp i:1000
0.24999992000001311	0.25000000000000006
321816.2755	330405.8918

So, more accurate for roughly the same effort or less.

2.46 Exercise 1.30

2.46.1 Question

The `sum` procedure above generates a linear recursion. The procedure can be rewritten so that the sum is performed iteratively. Show how to do this by filling in the missing expressions in the following definition:

2.46.2 Answer

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

Let's check the stats!

recursive	iterative
30051.080005	19568.685587

2.47 Exercise 1.31

2.47.1 Question A.1

The `sum` procedure is only the simplest of a vast number of similar abstractions that can be captured as higher-order procedures. Write an analogous procedure called `product` that returns the product of the values of a function at points over a given range.

2.47.2 Answer A.1

```
(define (product-iter term a next b)
  (define (iter a result)
    (if (> a b)
        result
        (iter (next a) (* result (term a)))))
  (iter a 1)) ;; start at 1 so it's not always 0
```

2.47.3 Question A.2

Show how to define `factorial` in terms of `product`.

2.47.4 Answer A.2

I was briefly stumped because `product` only counts upward. Then I realized that's just how it's presented and it can go either direction, since addition and multiplication are commutative. I look forward to building up a more intuitive sense of numbers.

```
<<product-iter>>
(define (identity x)
  x)
(define (inc x)
  (1+ x))

(define (factorial n)
  (product-iter identity 1 inc n))

(display (factorial 7))
```

2.47.5 Question A.3

Also use `product` to compute approximations to π using the formula

$$\frac{\pi}{4} = \frac{2 \cdot 4 \cdot 4 \cdot 6 \cdot 6 \cdot 8 \cdots}{3 \cdot 3 \cdot 5 \cdot 5 \cdot 7 \cdot 7 \cdots}$$

2.47.6 Answer A.3

Once this equation is encoded, you just need to multiply it by two to get π.

Fun fact: the formula is slightly wrong, it should start the series with $\frac{1}{2}$.

```
<<product-iter>>
(define (pi-product n)
  (define (div x)
```

```
    (let ((x1 (- x 1))
          (x2 (+ x 1)))
      (* (/ x x1) (/ x x2))))
  (* 2.0 (product-iter div 2 (lambda (z) (+ z 2)) n)))

(display (pi-product 100000))
```

```
3.1415769458228726
```

2.47.7 Question B

> If your product procedure generates a recursive process, write one that generates an iterative process. If it generates an iterative process, write one that generates a recursive process.

2.47.8 Answer B

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

```
<<mattbench2>>
<<print-table>>
<<product-iter>>
(define (pi-product n)
  (define (div x)
    (let ((x1 (- x 1))
          (x2 (+ x 1)))
      (* (/ x x1) (/ x x2))))
  (* 2.0 (product-iter div 2 (lambda (z) (+ z 2)) n)))
<<product-rec>>
(define (pi-product-rec n)
  (define (div x)
    (let ((x1 (- x 1))
          (x2 (+ x 1)))
      (* (/ x x1) (/ x x2))))
  (* 2.0 (product-rec div 2 (lambda (z) (+ z 2)) n)))

(define iterations 50000)
(print-table
 (list (list "iterative" "recursive")
       (list (cadr (mattbench2 (λ()(pi-product 1000)) iterations⌟
))
```

```
            (cadr (mattbench2 (λ()(pi-product-rec 1000)) iterations))))
 #:colnames #t)
```

iterative	recursive
1267118.0538	3067085.5323

2.48 Exercise 1.32

2.48.1 Question A

> Show that `sum` and `product` are both special cases of a still more general notion called `accumulate` that combines a collection of terms, using some general accumulation function:

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

> `accumulate` takes as arguments the same term and range specifications as `sum` and `product`, together with a `combiner` procedure (of two arguments) that specifies how the current term is to be combined with the accumulation of the preceding terms and a `null-value` that specifies what base value to use when the terms run out. Write `accumulate` and show how `sum` and `product` can both be defined as simple calls to `accumulate`.

2.48.2 Answer A

When I first did this question, I struggled a lot before realizing `accumulate` was much closer to the exact definitions of sum/product than I thought.

```
(define (accumulate-iter 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))
```

```
<<accumulate-iter>>

;; here you can see definitions in terms of accumulate
(define (sum term a next b)
  (accumulate-iter + 0 term a next b))
(define (product term a next b)
  (accumulate-iter * 1 term a next b))
```

```scheme

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

;; accumulate in action
(define (factorial n)
  (accumulate-iter * 1 identity 1 inc n))

(display (factorial 7))
```

```
5040
```

2.48.3 Question B

If your `accumulate` procedure generates a recursive process, write one that generates an iterative process. If it generates an iterative process, write one that generates a recursive process.

2.48.4 Answer B

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

2.49 Exercise 1.33

2.49.1 Question A

You can obtain an even more general version of `accumulate` by introducing the notion of a filter on the terms to be combined. That is, combine only those terms derived from values in the range that satisfy a specified condition. The resulting `filtered-accumulate` abstraction takes the same arguments as accumulate, together with an additional predicate of one argument that specifies the filter. Write `filtered-accumulate` as a procedure.

2.49.2 Answer A

```scheme
(define (filtered-accumulate-iter
           predicate? combiner null-value
           term a next b)
```

```scheme
  (define (iter a result)
    (cond ((> a b) result)
          ((predicate? a)
           (iter (next a)
                 (combiner result (term a))))
          (else (iter (next a)
                      result))))
  (iter a null-value))
```

2.49.3 Question B

Show how to express the following using `filtered-accumulate`:

1. A Find the sum of the squares of the prime numbers in the interval a to b (assuming that you have a `prime?` predicate already written)

```scheme
(load "mattcheck.scm")
(define (square x)
  (* x x))
<<filtered-accumulate-iter>>
<<expmod-mr2>>
<<mr-test2>>
<<mr-prime>>
(define mr-times 100)
(define (prime? x)
  (mr-prime? x mr-times))
(define (prime-sum a b)
  (filtered-accumulate-iter prime? + 0
                            square a 1+ b))

(mattcheck-equal "1 prime correct"
                 (prime-sum 1008 1010)
                 (square 1009)) ;; 1009
(mattcheck-equal "many primes correct"
                 (prime-sum 1000 2001)
                 (apply +
                        (map square
                             (filter prime? (iota (- 2001 1000)
                                                  1000)))))
```

```
SUCCEED at 1 prime correct
SUCCEED at many primes correct
```

2. B

Find the product of all the positive integers less than n that are relatively prime to n (i.e., all positive integers $i < n$ such that $\textsc{gcd}(i, n) = 1$.

```
(load "mattcheck.scm")
(define (square x)
  (* x x))
(define (id x) x)
<<filtered-accumulate-iter>>
<<gcd>>
(define (relative-prime? x y)
  (= 1 (gcd x y)))

(define (Ex_1-33B n)
  (filtered-accumulate-iter
   (λ(i) (relative-prime? i n))
   * 1 id
   1 1+ (1- n)))

(define (alternate n)
  (apply *
         (filter (λ(i) (relative-prime? i n))
                 (iota (- n 1) 1))))

(mattcheck-equal "Ex_1-33B"
                 (Ex_1-33B 100)
                 (alternate 100))
```

```
SUCCEED at Ex_1-33B
```

2.50 1.3.2: Constructing Procedures Using lambda

A procedure that's only used once is more conveniently expressed as the special form `lambda`.

Variables that are only briefly used in a limited scope can be specified with the special form `let`. Variables in `let` blocks override external variables. The authors recommend using `define` for procedures and `let` for variables.

2.51 Exercise 1.34

2.51.1 Question

Suppose we define the procedure

```scheme
(define (f g) (g 2))
```

Then, we have

```scheme
(f square)
; 4
(f (lambda (z) (* z (+ z 1))))
; 6
```

What happens if we (perversely) ask the interpreter to evaluate the combination `(f f)`? Explain.

2.51.2 Answer

It ends up trying to execute `2` as a function.

```scheme
;; Will be evaluated like this:
;;   (f f)
;;   (f 2)
;;   (2 2)
(define (f g) (g 2))
(f f)
```

```
ice-9/boot-9.scm:1685:16: In procedure raise-exception:
Wrong type to apply: 2
```

2.52 1.3.3 Procedures as General Methods

The **half-interval method**: if $f(a) < 0 < f(b)$, then f must have at least one 0 between a and b. To find 0, let x be the average of a and b, if $f(x) < 0$ then 0 must be between x and b, if $f(x) > 0$ than 0 must be between a and x.

The **fixed point** of a function satisfies the equation

$$f(x) = x$$

For some functions, we can locate a fixed point by beginning with an initial guess y and applying $f(y)$ repeatedly until the value doesn't change much.

Average damping can help converge fixed-point searches.

The symbol $\mapsto$ ("maps to") can be considered equivalent to a lambda. For example, $x \mapsto x + x$ is equivalent to `(lambda (x) (+ x x))`. In English, "the function whose value at y is x/y". *Though it seems like $\mapsto$ doesn't necessarily describe a function, but the value of a function at a certain point? Or maybe that would just be , ie $f(x)$ etc*

2.53 Exercise 1.35

2.53.1 Text

```scheme
(define (close-enough? x y)
  (< (abs (- x y)) 0.001))
```

```scheme
(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))
```

2.53.2 Question

Show that the golden ratio φ is a fixed point of the transformation $x \mapsto 1 + 1/x$, and use this fact to compute φ by means of the `fixed-point` procedure.

2.53.3 Answer

```scheme
<<close-enough>>
<<fixed-point-txt>>
(define golden-ratio
  (fixed-point (λ(x)(+ 1 (/ 1 x)))
               1.0))

(display golden-ratio)
```

```
1.6180327868852458
```

2.54 Exercise 1.36

2.54.1 Question

Modify `fixed-point` so that it prints the sequence of approximations it generates, using the `newline` and `display` primitives shown in Exercise 1.22. Then find a solution to $x^x = 1000$ by finding a fixed point of $x \mapsto \log(1000)/\log(x)$. (Use Scheme's primitive `log` procedure, which computes natural logarithms.) Compare the number of steps this takes with and without average damping. (Note that you cannot start `fixed-point` with a guess of 1, as this would cause division by $\log(1) = 0$.)

2.54.2 Answer

Using the `display` and `newline` functions at any great extent is pretty exhausting, so I'll use `format` instead.

```
(use-modules (ice-9 format))
(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)))
      (format #t "~&~a~%" next)
      (if (close-enough? guess next)
          next
          (try next))))
  (try first-guess))
```

```
<<close-enough>>
<<fixed-point-debug>>
(fixed-point (λ(x) (/ (log 1000) (log x))) 1.1)
```

Undamped, fixed-point makes 37 guesses.

```
<<close-enough>>
<<fixed-point-debug>>
(define (average x y)
  (/ (+ x y) 2))
(fixed-point (λ(x) (average (log x) (/ (log 1000) (log x)))) 1.1)
```

Damped, it makes 21.

2.55 Exercise 1.37

2.55.1 Question A

An infinite continued fraction is an expression of the form

$$f = \cfrac{N_1}{D_1 + \cfrac{N_2}{D_2 + \cfrac{N_3}{D_3 + \dots}}}$$

As an example, one can show that the infinite continued fraction expansion with the N_i and the D_i all equal to 1 produces $1/\varphi$, where φ is the golden ratio (described in 1.2.2). One way to approximate an infinite continued fraction is to truncate the expansion after a given number of terms. Such a truncation—a so-called k-term finite continued fraction}—has the form

$$\cfrac{N_1}{D_1 + \cfrac{N_2}{\ddots + \cfrac{N_k}{D_k}}}$$

Suppose that `n` and `d` are procedures of one argument (the term index i) that return the N_i and D_i of the terms of the continued fraction. Define a procedure `cont-frac` such that evaluating `(cont-frac n d k)` computes the value of the k-term finite continued fraction.

2.55.2 Answer A

A note: the "golden ratio" this code estimates is exactly `1.0` less than the golden ratio anyone else seems to be talking about.

```
(define (cont-frac n d k)
  (define (iter i result)
    (if (= i 0)
        result
        (iter (1- i) (/ (n i) (+ (d i) result)))))

  (iter (1- k) (/ (n k) (d k))))
```

2.55.3 Question B

Check your procedure by approximating $1/\varphi$ using

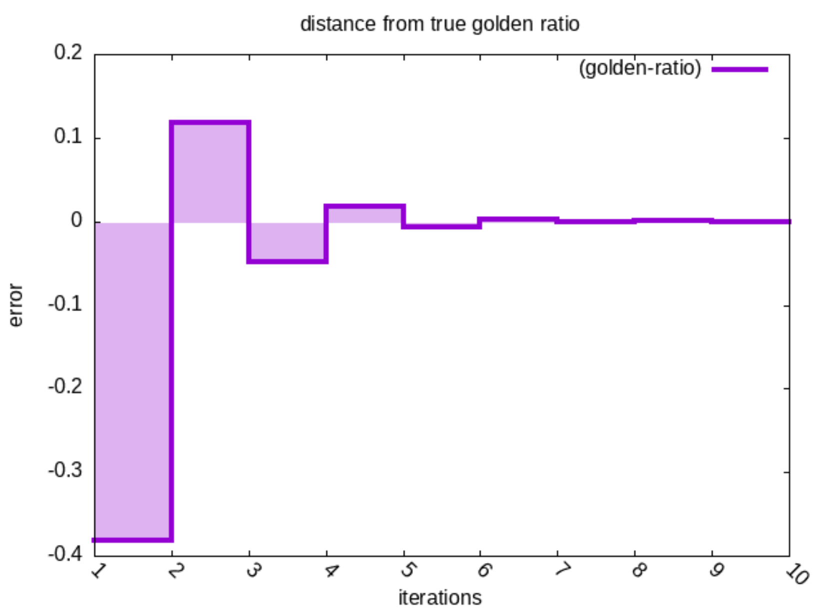
```
(cont-frac (lambda (i) 1.0)
           (lambda (i) 1.0)
```

```
        k)
```

for successive values of `k`. How large must you make `k` in order to get an approximation that is accurate to 4 decimal places?

2.55.4 Answer B

1	-0.3819660112501051
2	0.1180339887498949
3	-0.04863267791677173
4	0.018033988749894814
5	-0.0069660112501050975
6	0.0026493733652794837
7	-0.0010136302977241662
8	0.00038692992636546464
9	-0.00014782943192326314
10	5.6460660007306984e-05



k must be at least 10 to get precision of 4 decimal places.

2.55.5 Question C

If your `cont-frac` procedure generates a recursive process, write one that generates an iterative process. If it generates an iterative process, write one that generates a recursive process.

2.55.6 Answer C

```
(define (cont-frac-rec n d k)
  (define (rec i)
    (if (= i k)
        (/ (n i) (d i))
        (/ (n i) (+ (d i) (rec (1+ i))))))

  (rec 1))
```

```
<<cont-frac>>
<<cont-frac-rec>>
(define (golden-ratio k)
  (cont-frac (λ(i) 1.0)(λ(i)1.0) k))
(define (golden-ratio-rec k)
  (cont-frac-rec (λ(i) 1.0)(λ(i)1.0) k))

(load "mattcheck.scm")
(mattcheck-equal "cont-frac iter and recursive equivalence"
           (golden-ratio-rec 15)
           (golden-ratio 15))
```

```
SUCCEED at cont-frac iter and recursive equivalence
```

2.56 Exercise 1.38

2.56.1 Question

> In 1737, the Swiss mathematician Leonhard Euler published a memoir *De Fractionibus Continuis*, which included a continued fraction expansion for $e - 2$, where e is the base of the natural logarithms. In this fraction, the N_i are all 1, and the D_i are successively 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, ... Write a program that uses your `cont-frac` procedure from Exercise 1.37 to approximate e, based on Euler's expansion.

2.56.2 Answer

```
<<cont-frac>>
(define (euler k)
  (+ 2
     (cont-frac (λ(i) 1.0)
                (λ(i) (let ((j (1+ i)))
                        (if (= 0 (modulo j 3))
                            (* 2 (/ j 3))
```

```
                         1)))
              k)))

(euler 100)
```

```
2.7182818284590455
```

2.57 Exercise 1.39

2.57.1 Question

A continued fraction representation of the tangent function was published in 1770 by the German mathematician J.H. Lambert:

$$\tan x = \cfrac{x}{1 - \cfrac{x^2}{3 - \cfrac{x^2}{5 - \dots}}}$$

where x is in radians. Define a procedure `(tan-cf x k)` that computes an approximation to the tangent function based on Lambert's formula. `k` specifies the number of terms to compute, as in Exercise 1.37.

2.57.2 Answer

```
<<cont-frac>>
(define (tan-cf x k)
  (cont-frac (λ(i) (if (= i 1)
                       x
                       (* x x -1.0)))
             (λ(i) (if (= i 1)
                       1.0
                       (- (* i 2.0) 1.0)))
             k))

(tan-cf 55 101)
```

```
-45.1830879105221
```

2.58 1.3.4 Procedures as Returned Values

Procedures can return other procedures, which opens up new ways to express processes.

Newton's Method: $g(x) = 0$ is a fixed point of the function $x \mapsto f(x)$ where

$$f(x) = x - \frac{g(x)}{Dg(x)}$$

Where $x \mapsto g(x)$ is a differentiable function and $Dg(x)$ is the derivative of g evaluated at x.

2.59 Exercise 1.40

2.59.1 Text

```scheme
(define (average-damp f)
  (lambda (x) (average x (f x))))
```

```scheme
(define dx 0.00001)
```

```scheme
(define (deriv g)
  (lambda (x) (/ (- (g (+ x dx)) (g x)) dx)))
```

```scheme
(define (newton-transform g)
  (lambda (x) (- x (/ (g x) ((deriv g) x)))))
(define (newtons-method g guess)
  (fixed-point (newton-transform g) guess))
```

```scheme
<<average>>
<<average-damp>>
<<dx>>
<<deriv>>
<<newtons-method>>
```

2.59.2 Question

Define a procedure `cubic` that can be used together with the `newtons-method` procedure in expressions of the form:

```scheme
(newtons-method (cubic a b c) 1)
```

to approximate zeros of the cubic $x^3 + ax^2 + bx + c$.

2.59.3 Answer

```
(define (cubic a b c)
  (lambda (x)
    (+ (expt x 3)
       (* a (expt x 2))
       (* b x)
       c)))
```

```
(define (cubic-zero a b c)
  (newtons-method (cubic a b c) 1))
```

```
<<fixed-point-txt>>
<<newtons-method-txt>>
<<cubic>>
<<cubic-zero>>

(cubic-zero 2 3 4)
```

2.60 Exercise 1.41

2.60.1 Question

> Define a procedure `double` that takes a procedure of one argument as argument and returns a procedure that applies the original procedure twice. For example, if `inc` is a procedure that adds 1 to its argument, then `(double inc)` should be a procedure that adds 2. What value is returned by

```
(((double (double double)) inc) 5)
```

2.60.2 Answer

```
(define (double f)
  (λ (x)
    (f (f x))))
```

```
(define inc 1+)
<<double>>
<<Ex1-41>>
```

```
21
```

2.61 Exercise 1.42

2.61.1 Question

Let f and g be two one-argument functions. The composition f after g is defined to be the function $x \mapsto f(g(x))$. Define a procedure `compose` that implements composition.

2.61.2 Answer

```
(define (compose f g)
  (λ(x)
    (f (g x))))
```

```
<<compose>>
<<square>>
(define inc 1+)
((compose square inc) 6)
```

```
49
```

2.62 Exercise 1.43

2.62.1 Question

If f is a numerical function and n is a positive integer, then we can form the n^{th} repeated application of f, which is defined to be the function whose value at x is $f(f(\ldots(f(x))\ldots))$. For example, if f is the function $x \mapsto x+1$, then the n^{th} repeated application of f is the function $x \mapsto x+n$. If f is the operation of squaring a number, then the n^{th} repeated application of f is the function that raises its argument to the 2^n-th power. Write a procedure that takes as inputs a procedure that computes f and a positive integer n and returns the procedure that computes the n^{th} repeated application of f.

2.62.2 Answer

```
<<compose>>
(define (repeated f n)
  (if (= n 1)
      f
      (repeated (compose f f)
                (- n 1))))
```

```
<<square>>
<<repeated>>
(if (= ((repeated square 2) 5) 625)
    "Success"
    "Fail")
```

```
Success
```

2.63 Exercise 1.44

2.63.1 Question

> The idea of smoothing a function is an important concept in signal processing. If f is a function and dx is some small number, then the smoothed version of f is the function whose value at a point x is the average of $f(x - dx)$, $f(x)$, and $f(x + dx)$. Write a procedure `smooth` that takes as input a procedure that computes f and returns a procedure that computes the smoothed f. It is sometimes valuable to repeatedly smooth a function (that is, smooth the smoothed function, and so on) to obtain the n-fold smoothed function. Show how to generate the n-fold smoothed function of any given function using `smooth` and `repeated` from Exercise 1.43.

2.63.2 Answer

```
<<average-varargs>>
(define (smooth f)
  (λ(x)
    (average (f (- x dx))
             (f x)
             (f (+ x dx)))))
(define (smooth-n f n)
  ((repeated smooth n) f))
```

2.64 Exercise 1.45

2.64.1 Question

> We saw in 1.3.3 that attempting to compute square roots by naively finding a fixed point of $y \mapsto x/y$ does not converge, and that this can be fixed by average damping. The same method works for finding cube roots as fixed points of the average-damped $y \mapsto x/y^2$. Unfortunately, the process does not work for fourth roots—a single average damp is not enough to make a fixed-point search for $y \mapsto x/y^3$ converge. On the other hand, if we average damp twice (i.e., use the average damp of the average damp of $y \mapsto x/y^3$) the fixed-point

search does converge. Do some experiments to determine how many average damps are required to compute n^{th} roots as a fixed-point search based upon repeated average damping of $y \mapsto x/y^{n-1}$. Use this to implement a simple procedure for computing n^{th} roots using `fixed-point`, `average-damp`, and the `repeated` procedure of Exercise 1.43. Assume that any arithmetic operations you need are available as primitives.

2.64.2 Answer

So this is strange. Back in my original workthrough of this book, I'd decided that finding an nth root required $\lfloor\sqrt{n}\rfloor$ dampings. With a solution like this:

```
<<fixed-point-txt>>
<<repeated>>
<<average-damp>>
(define (sqrt n)
  (fixed-point
   (average-damp
    (lambda (y)
      (/ x y)))
   1.0))
(define (nth-root x n)
  (fixed-point
   ((repeated average-damp (ceiling (sqrt n)))
    (lambda (y)
      (/ x (expt y (- n 1)))))
   1.0))
```

While this solution appears to work fine, my experiments are suggesting that it takes *less* than $\lfloor\sqrt{n}\rfloor$. For example, I originally thought powers of 16 required four dampings, but this code isn't failing until it reaches powers of 32.

```
;; Version of ”repeated” that can handle being asked to repeat zero times.
<<compose>>
<<identity>>
(define (repeated f n)
  (define (rec m)
  (if (= n 1)
      f
      (repeated (compose f f)
                (- n 1))))
  (if (= n 0)
      identity
```

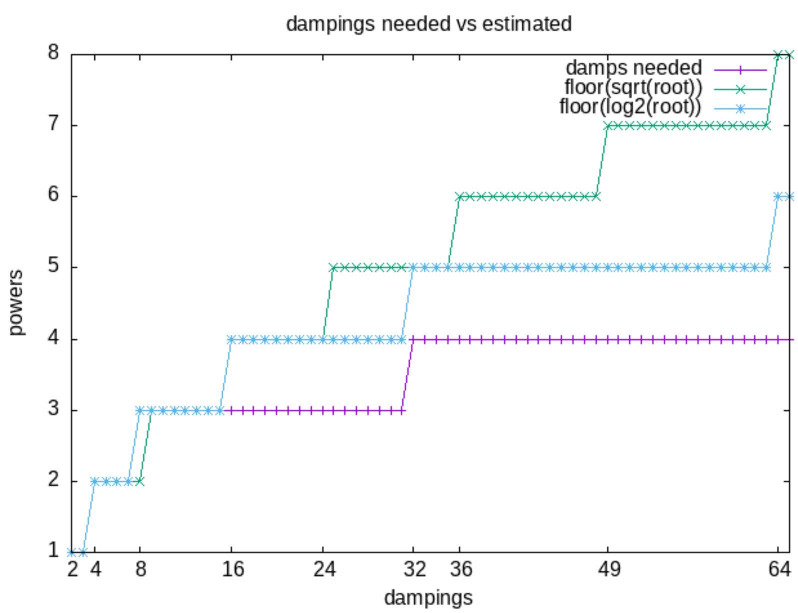
```
      (rec n)))
```

```
;; version of ”fixed-point” that will give up after a certain number of guesses.
(define (limited-fixed-point f first-guess)
  (define limit 5000)
  (define tolerance 0.00000001)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2))
       tolerance))
  (define (try guess tries)
    (if (= tries limit)
        ”LIMIT REACHED”
        (let ((next (f guess)))
          (if (close-enough? guess next)
              next
              (try next (+ 1 tries))))))
    (try first-guess 1))
```

Let’s automatically find how many dampings are necessary. We can make a program that finds higher and higher nth roots, and adds another layer of damping when it hits the error. It returns a list of nth roots along with how many dampings were needed to find them.

```
<<fixed-point-txt>>
<<limited-fixed-point>>
<<repeated>>
<<average-damp>>
<<average>>
<<print-table>>
(define (sqrt x)
  (fixed-point
   (average-damp
    (lambda (y) (/ x y)))
   1.0))
(define (nth-tester base n-max)
  (define (iter ll)
    (let ((n (+ 2 (length ll))))
      (define (try damps)
        (let ((x (limited-fixed-point
                  ((repeated average-damp damps)
                   (lambda (y)
                     (/ base (expt y (- n 1)))))
                  1.1)))
```

```
          (if (string? x)
              (try (1+ damps))
              (list base n x damps))))
      (if (> n n-max)
          ll
          (iter (cons (try 1) ll)))))

  (iter '()))
(let* ((t (reverse (nth-tester 3 65))))
  (cons '("root" "result" "damps needed" "floor(sqrt(root))" "floor(log2(root))")
        (map (λ(x)
               (append x
                       (list (floor (sqrt (car x)))
                             (floor (/ (log (car x))(log 2))))))
             (map cdr t))))
```



I've spent too much time on this problem already but I have to wonder about floating-point issues, given that they are the core of the `good-enough` procedure. I have to wonder whether a `fixed-point` version that replaces the `tolerance` decision making, and instead retains the last three guesses and checks for a loop.

2.65 Exercise 1.46

2.65.1 Question

Several of the numerical methods described in this chapter are instances of an extremely general computational strategy known as *iterative improvement*. Iterative improvement says that, to compute something, we start with an initial guess for the answer, test if the guess is good enough, and otherwise improve the guess and continue the process using the improved guess as the new guess. Write a procedure `iterative-improve` that takes two procedures as arguments: a method for telling whether a guess is good enough and a method for improving a guess. `iterative-improve` should return as its value a procedure that takes a guess as argument and keeps improving the guess until it is good enough. Rewrite the `sqrt` procedure of 1.1.7 and the `fixed-point` procedure of 1.3.3 in terms of `iterative-improve`.

2.65.2 Answer

```
(define (iterative-improve good-enough? improve)
  (λ(first-guess)
    (define (iter guess)
      (let ((next (improve guess)))
        (if (good-enough? guess next)
            next
            (iter next))))
    (iter first-guess)))
```

```
<<iterative-improve>>
(define tolerance 0.00001)

(define (fixed-point-improve f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2))
       tolerance))
  ((iterative-improve close-enough? f) first-guess))
```

```
<<average>>
<<iterative-improve>>
(define (improve guess x)
  (average guess (/ x guess)))
(define (good-enough? guess x)
   (= (improve guess x) guess))
```

```
(define (sqrt-improve x)
  ((iterative-improve
    (λ(guess) (improve guess x))
    (λ(guess) (good-enough? guess x)))
   1.0))
```

```
(load "mattcheck2.scm")
<<fixed-point-txt>>
<<fixed-point-improve>>
(mattcheck "fixed-point-improve still working"
                 (fixed-point (λ(x)(+ 1 (/ 1 x))) 1.0)
                 (fixed-point-improve (λ(x)(+ 1 (/ 1 x))) 1.0))
<<sqrt>>
<<sqrt-improve>>
(mattcheck "sqrt-improve still working"
                 (sqrt 5)
                 (sqrt 5))
```

```
SUCCEED at fixed-point-improve still working
SUCCEED at sqrt-improve still working
```

3 Chapter 2: Building Abstractions with Data

The basic representations of data we've used so far aren't enough to deal with complex, real-world phenomena. We need to combine these representations to form **compound data**.

The technique of isolating how data objects are *represented* from how they are *used* is called **data abstraction**.

3.1 2.1.1: Example: Arithmetic Operations for Rational Numbers

Lisp gives the procedures `cons`, `car`, and `cdr` to create **pairs**. This is an easy system for representing rational numbers.

Note that the system proposed for representing and working with rational numbers has **abstraction barriers** isolating different parts of the system. The parts that use rational numbers don't know how the constructors and selectors for rational numbers work, and the constructors and selectors use the underlying Lisp interpreter's pair functions without caring how they work.

Note that these abstraction layers allow the developer to change the underlying architecture without modifying the programs that depend on it.

3.2 Exercise 2.1

3.2.1 Question

> Define a better version of `make-rat` that handles both positive and negative arguments. `make-rat` should normalize the sign so that if the rational number is positive, both the numerator and denominator are positive, and if the rational number is negative, only the numerator is negative.

3.2.2 Answer

3.3 Exercise 2.2

3.3.1 Question

Consider the problem of representing line segments in a plane. Each segment is represented as a pair of points: a starting point and an ending point. Define a constructor `make-segment` and selectors `start-segment` and `end-segment` that define the representation of segments in terms of points. Furthermore, a point can be represented as a pair of numbers: the x coordinate and the y coordinate. Accordingly, specify a constructor `make-point` and selectors `x-point` and `y-point` that define this representation. Finally, using your selectors and constructors, define a procedure `midpoint-segment` that takes a line segment as argument and returns its midpoint (the point whose coordinates are the average of the coordinates of the endpoints). To try your procedures, you'll need a way to print points:

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

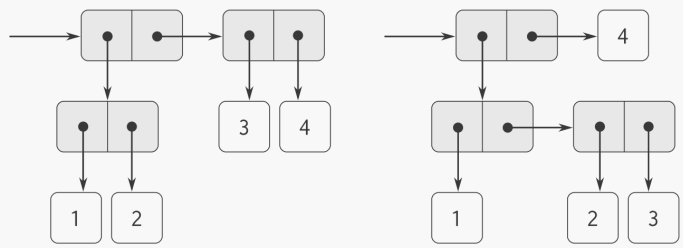
3.3.2 Answer

3.4 2.1.3: What Is Meant by Data?

We can consider data as being a collection of selectors and constructors, together with specific conditions that these procedures must fulfill in order to be a valid representation. For example, in the case of our rational number implementation, for rational number x made with numerator n and denominator d, dividing the result of `(numer x)` over the result of `(denom x)` should be equivalent to dividing n over d.

3.5 2.2: Hierarchical Data and the Closure Property

`cons` pairs can be used to construct more complex data-types.



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

Figure 2.3: Two ways to combine 1, 2, 3, and 4 using pairs.

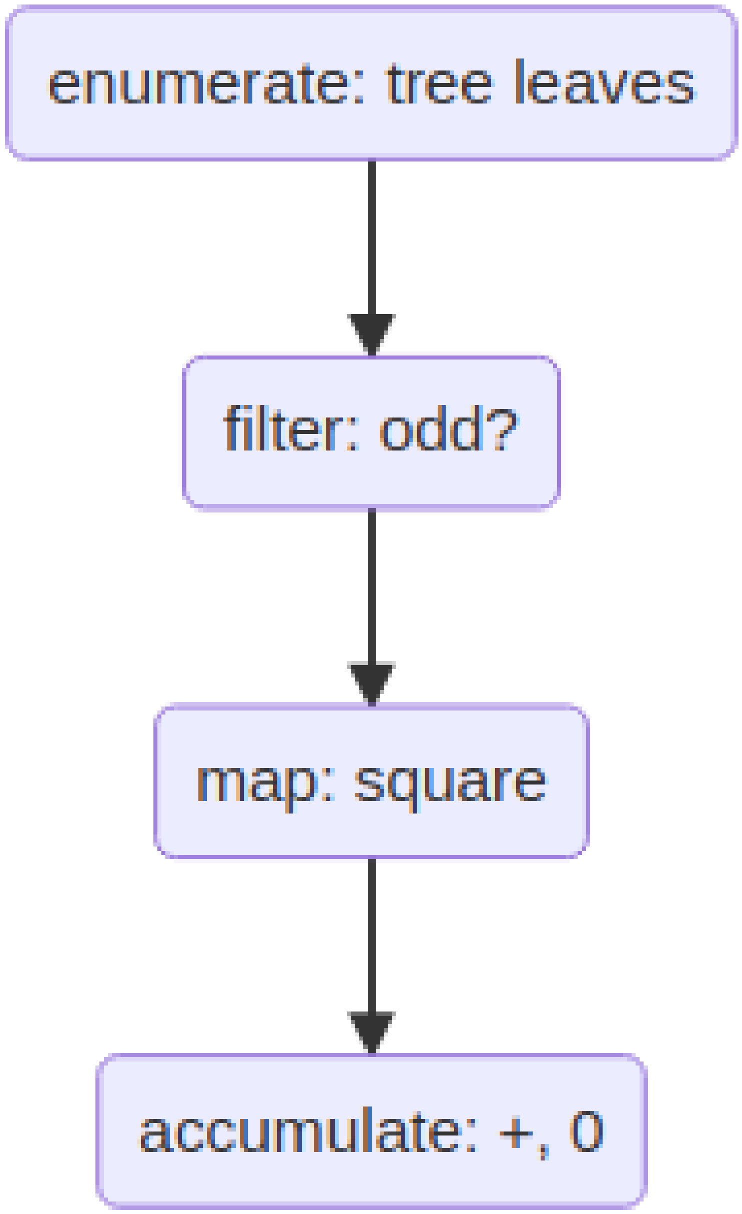
The ability to combine things using an operation, then combine those results using the same operation, can be called the **closure property**. `cons` can create pairs whose elements are pairs, which satisfies the closure property. This property enables you to create hierarchical structures. We've already regularly used the closure property in creating procedures composed of other procedures.

> **Definitions of "closure"**
>
> The use of the word "closure" here comes from abstract algebra, where a set of elements is said to be closed under an operation if applying the operation to elements in the set produces an element that is again an element of the set. The Lisp community also (unfortunately) uses the word "closure" to describe a totally unrelated concept: A closure is an implementation technique for representing procedures with free variables. We do not use the word "closure" in this second sense in this book.

3.6 2.2.3: Sequences as Conventional Interfaces

Abstractions are an important part of making code clearer and more easy to understand. One beneficial manner of abstraction is making available conventional interfaces for working with compound data, such as `filter` and `map`.

This allows for easily making "signal-flow" conceptions of processes:


enumerate: tree leaves
filter: odd?
map: square
accumulate: +, 0

3.7 2.2.4: Example: A Picture Language

Authors describe a possible implementation of a “picture language” that tiles, patterns, and warps images according to a specification. This language consists of:

- a **painter** which makes an image within a specified parallelogram shaped frame. This is the most primitive element.
- **Operations** which make new painters from other painters. For example:
  - *beside* takes two painters, producing a new painter that puts one in the left half and one in the right half.
  - *flip-horiz* takes one painter and produces another to draw its image right-to-left reversed. These are defined as Scheme procedures and therefore have all the properties of Scheme procedures.