A Journey Through SICP

Notes, exercises and analyses of Abelson and Sussman

ProducerMatt

March 30, 2023

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 (**lambda**(**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.ă(**+ 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:
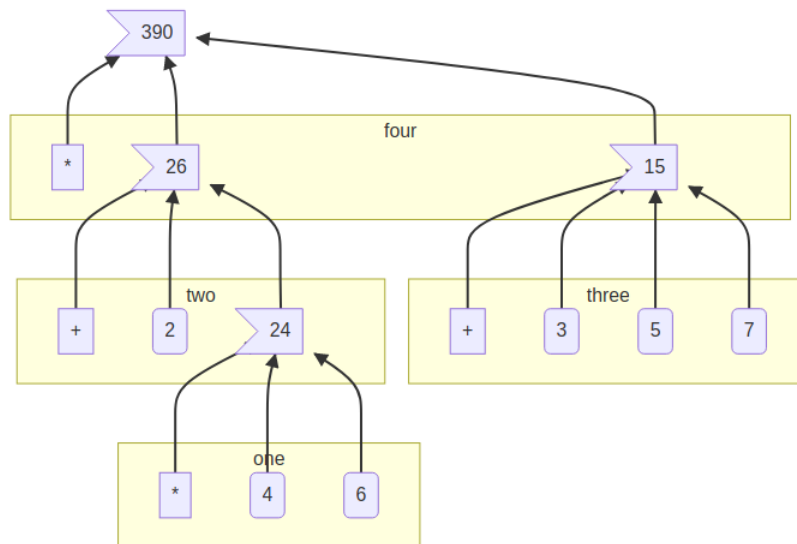
2.4   1.1.4: Compound Procedures

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.

```
1   (define (square x) (* x x)) ; to square something, multiply it by itself
2   ; now it can be applied or used in other definitions:
3   (square 4) ; ⇒ 16
4
5   (define (sum-of-squares x y)
6     (+ (square x) (square y)))
7   (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.

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

```
1   ; Applicative evaluation
2   (f 5)
3   (sum-of-squares (+ a 1) (* a 2))
4   (sum-of-squares (+ 5 1) (* 5 2))
5   (sum-of-squares 6 10)
6   (+ (square x)(square y))
7   (+ (square 6)(square 10))
8   (+ (* 6 6)(* 10 10))
9   (+ 36 100)
10  136
11  ; Normal evaluation
12  (f 5)
13  (sum-of-squares (+ a 1) (* a 2))
14  (sum-of-squares (+ 5 1) (* 5 2))
15  (+ (square (+ 5 1)) (square (* 5 2)))
16  (+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2)))
17  (+ (* 6 6) (* 10 10))
18  (+ 36 100)
```

```
19    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**.

```
1  (cond
2      (predicate1 consequent1)
3      ...
4      (predicateN consequentN))
```

A shorter form of conditional:

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

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

```
1  (and expression1 ... expressionN)
2  ; if encounters false, stop eval and returns false.
3  (or expression1 ... expressionN)
4  ; if encounters true, stop eval and return true. Else false.
5  (not expression)
6  ; true is expression is false, false if expression is true.
```

A small clarification:

```
1  (define A (* 5 5))
2  (define (D) (* 5 5))
3  A ;  ⇒ 25
4  D ;  ⇒ compound procedure D
5  (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: Trying expressions

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

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

2.8 Exercise 1.2: Prefix form

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

```
1   (/ (+ 5 2 (- 2 3 (+ 6 (/ 4 5))))
2      (* 3 (- 6 2) (- 2 7)))
```

1/75

2.9 Exercise 1.3: Conditionals

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

8

```
1  <<square>>
2  (define (sum-square x y)
3    (+ (square x) (square y)))
4  (define (square-2of3 a b c)
5    (cond ((and ( ⩾ a b) ( ⩾ b c)) (sum-square a b))
6          ((and ( ⩾ a b) (> c b)) (sum-square a c))
7          (else (sum-square b c))))
```

```
1  <<EX1-3>>
2  <<try-these>>
3  (try-these square-2of3 '(7 5 3)
4                         '(7 3 5)
5                         '(3 5 7))
```

$$
\begin{array}{ll}
(7\ 5\ 3) & 74 \\
(7\ 3\ 5) & 74 \\
(3\ 5\ 7) & 74
\end{array}
$$

2.10   Exercise 1.4: Compound expressions

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:

```
1  (define (a-plus-abs-b a b)
2    ((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:

```
1  if b > 0: a + b
2  else: a - b
```

2.11   Exercise 1.5: Applicative vs normal-order evaluation

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:

9

```
1  (define (p) (p))
2
3  (define (test x y)
4    (if (= x 0)
5        0
6        y))
```

Then he evaluates the expression:

```
1  (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: Special form evaluation

2.14.1 Text code

```
1  (define (abs x)
2    (if (< x 0)
3        (- x)
4        x))
```

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

```
1  <<average>>
2  (define (improve guess x)
3    (average guess (/ x guess)))
4
5  <<square>>
6  <<abs>>
7  (define (good-enough? guess x)
8    (< (abs (- (square guess) x)) 0.001))
9
10 (define (sqrt-iter guess x)
11   (if (good-enough? guess x)
12       guess
13       (sqrt-iter (improve guess x) x)))
14
15 (define (sqrt x)
16   (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**:

```
1  (define (new-if predicate
2                  then-clause
3                  else-clause)
4    (cond (predicate then-clause)
```

```
5            (else else-clause)))
```

Eva demonstrates the program for Alyssa:

```
1   (new-if (= 2 3) 0 5)
2   ;;  ⇒ 5
3
4   (new-if (= 1 1) 0 5)
5   ;;  ⇒ 0
```

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

```
1   (define (sqrt-iter guess x)
2     (new-if (good-enough? guess x)
3             guess
4             (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 `s⌋qrt-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: `sqrt` with small and large numbers

2.15.1   Text

```
1   (define (mean-square x y)
2     (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.

```
1   <<txt-sqrt>>
2   (define (inferior-good-enough? guess lastguess)
3     (≤
4       (abs (-
5             (/ lastguess guess)
6             1))
7       0.0000000000001)) ; dx
8   (define (new-sqrt-iter guess x lastguess) ;; Memory of previous
    ↪  value
9     (if (inferior-good-enough? guess lastguess)
10         guess
11         (new-sqrt-iter (improve guess x) x guess)))
12  (define (new-sqrt x)
13    (new-sqrt-iter 1.0 x 0))
```

This solution can correctly find small and large numbers:

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

```
3162277.6601683795
```

```
1   <<try-these>>
2   <<inferior-good-enough>>
3   (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:

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

2. Imroving `sqrt` by avoiding extra `improve` call

   (a) Non-optimized

```
1    (use-modules (ice-9 format))
2    (load "../mattbench.scm")
3    (define (average x y)
4      (/ (+ x y) 2))
5    (define (improve guess x)
6      (average guess (/ x guess)))
7    (define (good-enough? guess x)
8       (= (improve guess x) guess)) ;; improve call 1
9    (define (sqrt-iter guess x)
10     (if (good-enough? guess x)
11         guess
12         (sqrt-iter (improve guess x) x))) ;; call 2
13   (define (sqrt x)
14     (sqrt-iter 1.0 x))
15   (newline)
16   (display (mattbench (lambda() (sqrt 69420)) 400000000))
17   (newline)
18   ;; 4731.30 <- Benchmark results
```

(b) Optimized

```
1    (use-modules (ice-9 format))
2    (load "../mattbench.scm")
3    (define (average x y)
4      (/ (+ x y) 2))
5    (define (improve guess x)
6      (average guess (/ x guess)))
7    (define (good-enough? guess nextguess x)
8      (= nextguess guess))
9    (define (sqrt-iter guess x)
10     (let ((nextguess (improve guess x)))
11       (if (good-enough? guess nextguess x)
12           guess
13           (sqrt-iter nextguess x))))
14   (define (sqrt x)
15     (sqrt-iter 1.0 x))
16   (newline)
17   (display (mattbench (lambda() (sqrt 69420)) 400000000))
18   (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):

```
1  <<txt-sqrt>>
2  <<try-these>>
3  (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.

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

1000000.0

So, my definition of `sqrt`:

```
1  <<average>>
2  (define (improve guess x)
3    (average guess (/ x guess)))
4  (define (good-enough? guess x)
5     (= (improve guess x) guess))
6  (define (sqrt-iter guess x)
7    (if (good-enough? guess x)
8         guess
9         (sqrt-iter (improve guess x) x)))
10 (define (sqrt x)
11   (sqrt-iter 1.0 x))
```

```
1  <<try-these>>
2  <<sqrt>>
3  (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: Cube roots

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

$$\frac{\frac{x}{y^2} + 2y}{3}$$

Use this formula to implement a cube-root procedure analogous to the `squ‿are-root` procedure. (In 1.3.4 Procedures as Returned Values 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:

```
1   <<square>>
2   <<try-these>>
3   (define (cb-good-enough? guess x)
4     (= (cb-improve guess x) guess))
5   (define (cb-improve guess x)
6     (/
7       (+
8         (/ x (square guess))
9         (* guess 2))
10      3))
11  (define (cbrt-iter guess x counter)
12    (if (or (cb-good-enough? guess x) (> counter 100))
13        guess
14        (begin
15          (cbrt-iter (cb-improve guess x) x (+ 1 counter)))))
16  (define (cbrt x)
17    (cbrt-iter 1.0 x 0))
18
19  (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

```
1   <<square>>
2   (define (cb-good-enough? nextguess guess lastguess x)
3     (or (= nextguess guess)
4         (= nextguess lastguess)))
5   (define (cb-improve guess x)
6     (/
7      (+
8       (/ x (square guess))
9       (* guess 2))
10     3))
11  (define (cbrt-iter guess lastguess x)
12    (define nextguess (cb-improve guess x))
13    (if (cb-good-enough? nextguess guess lastguess x)
14        nextguess
15        (cbrt-iter nextguess guess x)))
16  (define (cbrt x)
17    (cbrt-iter 1.1 9999 x))
```

```
1   <<cbrt>>
2   <<try-these>>
3   (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:

```
1   (define (factorial-recursion n)
2     (if (= n 1)
3         1
4         (* n
5            (factorial-recursion (- n 1)))))
6
7   (define (factorial-iteration n)
8     (define (fact-iter product counter max-count)
9       (if (> counter max-count)
10           product
```

```
11              (fact-iter
12                        (* counter product)
13                        (+ counter 1)
14                        max-count)))
15
16       (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: Peano counting and recursion

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.

```
1    (define (+ a b)
2      (if (= a 0)
3          b
4          (inc (+ (dec a) b))))
5
6    (define (+ a b)
7      (if (= a 0)
8          b
9          (+ (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

```
1   (+ 4 5)
2   (inc (+ 3 5))
3   (inc (inc (+ 2 5)))
4   (inc (inc (inc (+ 1 5))))
5   (inc (inc (inc (inc (+ 0 5)))))
6   (inc (inc (inc (inc 5))))
7   (inc (inc (inc 6)))
8   (inc (inc 7))
9   (inc 8)
10  9
```

2. iterative procedure

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

2.20 Exercise 1.10: Ackermann's Function

2.20.1 Question

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

```
1   (define (A x y)
2     (cond ((= y 0) 0)
3           ((= x 0) (* 2 y))
4           ((= y 1) 2)
5           (else (A (- x 1)
6                    (A x (- y 1))))))
```

What are the values of the following expressions?

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

$$
\begin{array}{ll}
(1\ 10) & 1024 \\
(2\ 4) & 65536 \\
(3\ 3) & 65536
\end{array}
$$

```
1  <<ackermann>>
2  (define (f n) (A 0 n))
3  (define (g n) (A 1 n))
4  (define (h n) (A 2 n))
5  (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

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

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

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

$$
\begin{array}{cc}
1 & 2 \\
2 & 4 \\
3 & 16 \\
4 & 65536
\end{array}
$$

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:

```
1    (define (fib n)
2      (cond ((= n 0) 0)
3            ((= n 1) 1)
4            (else (+ (fib (- n 1))
5                     (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.

```
1    (define (fib n)
2      (fib-iter 1 0 n))
3    (define (fib-iter a b count)
4      (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: More recursion vs iteration

2.22.1   Question

A function f is defined by the rule that:

$$f(n) = n \text{ if } n < 3$$

and

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

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

```
1   (define (fr n)
2     (if (< n 3)
3         n
4         (+       (fr (- n 1))
5             (* 2 (fr (- n 2)))
6             (* 3 (fr (- n 3)))))))
```

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

1	1
3	4
5	25
10	1892

2. Iterative

    (a) Attempt 1

```
1   ;; This seems like it could be better
2   (define (fi n)
3     (define (formula l)
4       (let ((a (car l))
5             (b (cadr l))
6             (c (caddr l)))
7         (+ a
```

22

```
 8              (* 2 b)
 9              (* 3 c))))
10      (define (iter l i)
11        (if (= i n)
12            (car l)
13            (iter (cons (formula l) l)
14                  (+ 1 i))))
15      (if (< n 3)
16          n
17          (iter '(2 1 0) 2)))
```

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

1	1
3	4
5	25
10	1892

It works but it seems wasteful.

(b) Attempt 2

```
 1  (define (fi2 n)
 2    (define (formula a b c)
 3        (+ a
 4           (* 2 b)
 5           (* 3 c)))
 6    (define (iter a b c i)
 7      (if (= i n)
 8          a
 9          (iter (formula a b c)
10                a
11                b
12                (+ 1 i))))
13    (if (< n 3)
14        n
15        (iter 2 1 0 2)))
```

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

1	1
3	4
5	25
10	1892

I like that better.

23

2.23 Exercise 1.12: Pascal's Triangle

2.23.1 Question

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

```
        1
      1   1
    1   2   1
  1   3   3   1
1   4   6   4   1
      .  .  .
```

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 corner of an infinite spreadsheet.

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

```
<<try-these>>
<<pascal-rec>>
(let ((l (iota 8)))
  (map (lambda (row)
         (map (lambda (xy)
                (apply pascal xy))
              row))
       (map (lambda (x)
              (map (lambda (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: Proving Fibonacci approximation optional

2.24.1 Question

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

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

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

```
(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

```
<<sqrt>>
(define sqrt5
  (sqrt 5))
(define phi
  (/ (+ 1 sqrt5) 2))
```

```
6    (define upsilon
7      (/ (- 1 sqrt5) 2))
8    (define (fib-phi n)
9      (/ (- (expt phi n)
10            (expt upsilon n))
11          sqrt5))
```

```
1    (use-srfis '(1))
2    <<fib-iter>>
3    <<fib-phi>>
4    <<try-these>>
5
6    (let* ((vals (drop (iota 21) 10))
7           (fibs (map fib-iter vals))
8           (approx (map fib-phi vals)))
9      (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.

Fib(n) minus Fib-phi(n)
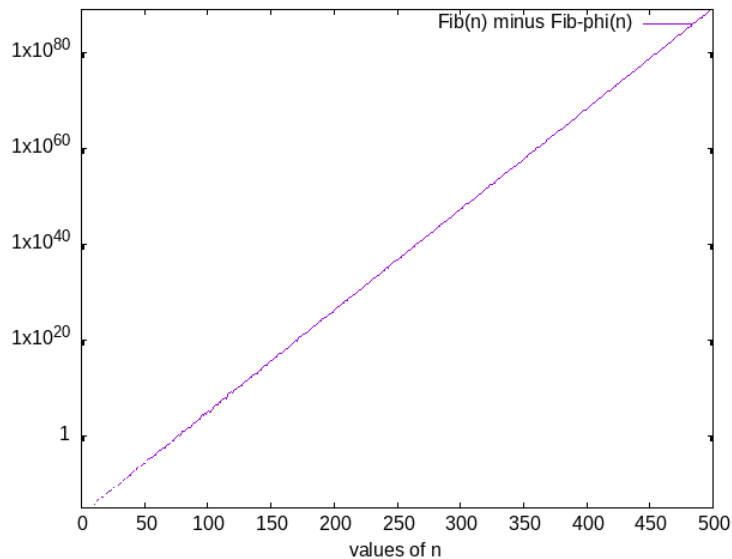
2.25 1.2.3: Orders of Growth

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: `count-change`

2.26.1 Text

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)))
```

2.26.2   Question A: Draw the tree

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

2.26.3   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 "\"")
```

```
19                     (display " -> ") ;; arrow pointing from parent to
                          ↪  child
20                     (display "\"") ;; child bubble
21                     (display `(,bubblecounter ,new-amount ,new-kinds))
22                     (display "\"")
23                     (display "\n")
24                     (cc new-amount new-kinds bubblecounter)))))
25       (set! bubblecounter (+ bubblecounter 1))
26       (cond ((= amount 0) 1)
27             ((or (< amount 0) (= kinds-of-coins 0)) 0)
28             (else (+
29                    (recur amount (- kinds-of-coins 1))
30                    (recur (- amount
31                             (first-denomination kinds-of-coins))
32                           kinds-of-coins))))))

34  (define (first-denomination kinds-of-coins)
35     (cond ((= kinds-of-coins 1) 1)
36           ((= kinds-of-coins 2) 5)
37           ((= kinds-of-coins 3) 10)
38           ((= kinds-of-coins 4) 25)
39           ((= 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**.
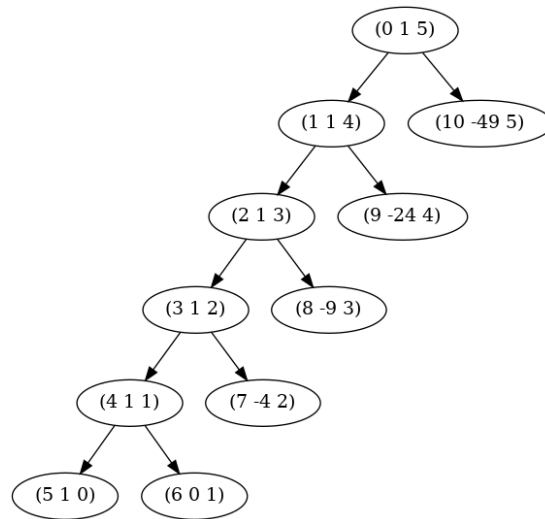
```
1  <<count-change-graphviz>>
2  (count-change 1)
```
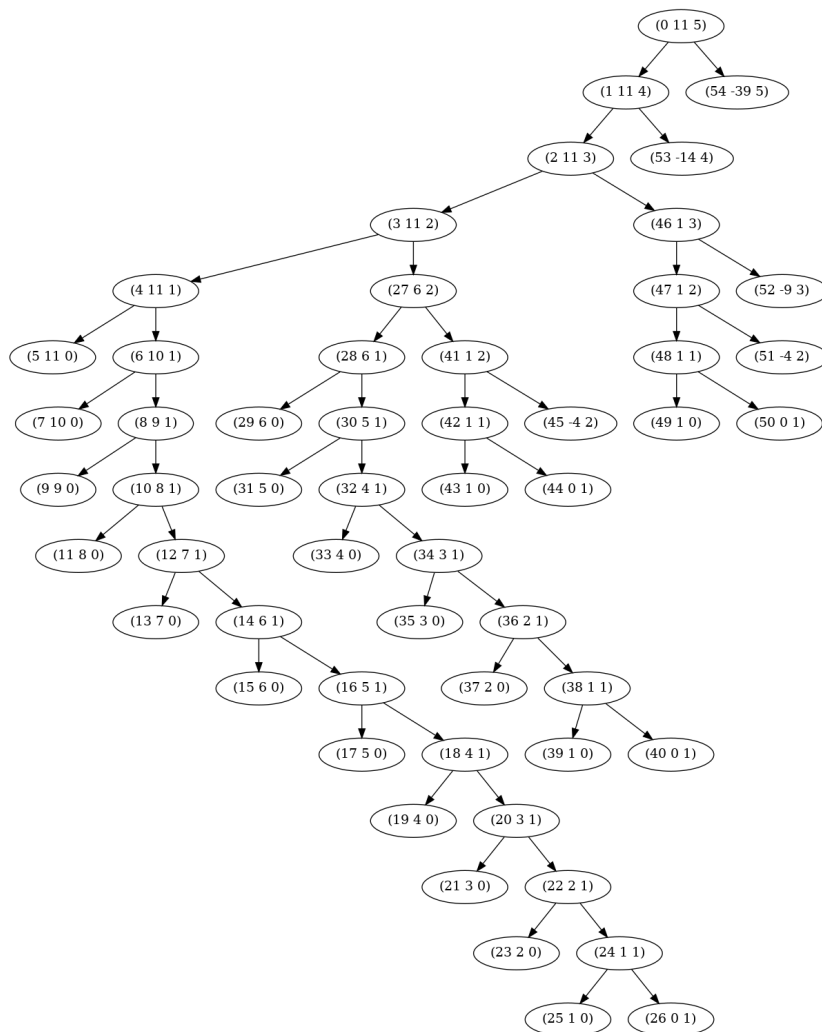
```
1  digraph {
2  "(0 1 5)" -> "(1 1 4)"
3  "(1 1 4)" -> "(2 1 3)"
4  "(2 1 3)" -> "(3 1 2)"
5  "(3 1 2)" -> "(4 1 1)"
6  "(4 1 1)" -> "(5 1 0)"
7  "(4 1 1)" -> "(6 0 1)"
8  "(3 1 2)" -> "(7 -4 2)"
9  "(2 1 3)" -> "(8 -9 3)"
10 "(1 1 4)" -> "(9 -24 4)"
11 "(0 1 5)" -> "(10 -49 5)"
12 }
```

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

2.26.4   Question B: Analyzing process growth

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

2.26.5   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)))
```
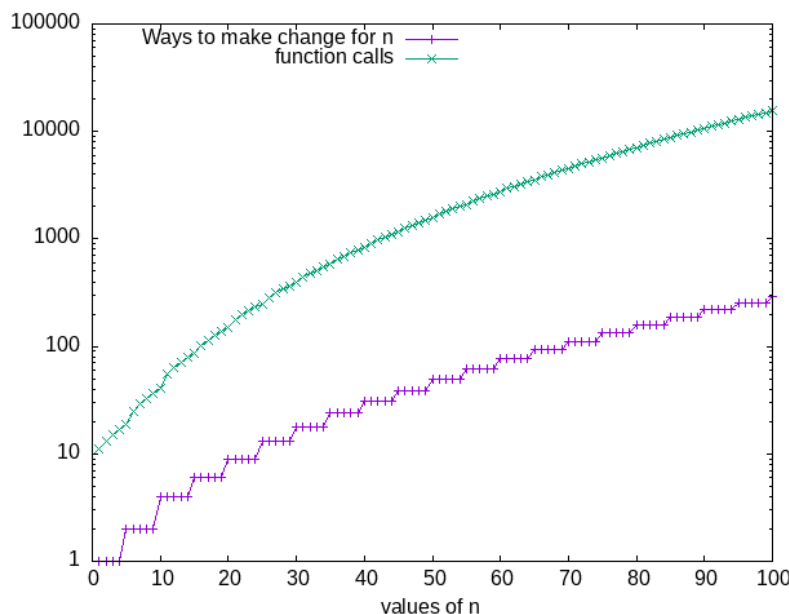
I believe the space to be $\Theta(n + d)$ as the function calls count down the denominations before counting down the change. However I notice most answers describe $\Theta(n)$ instead, maybe I'm being overly pedantic and getting the wrong answer.

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:

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

33

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[1]. 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: Sine approximation

2.27.1 Question A

The sine of an angle (specified in radians) can be computed by making use of the approximation $\sin x \approx x$ 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!

---

[1] https://codology.net/post/sicp-solution-exercise-1-14/

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

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

$$-0.39980345741334 \quad 5$$

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

```
1  (use-srfis '(1))
2  <<1-15-p-measure>>
3  (let* ((vals (iota 300 0.1 0.1))
4         (sines (map (lambda (i)
5                       (cdr (sine i)))
6                     vals)))
7    (zip vals sines))
```

```
1  (use-srfis '(1))
2  <<1-15-p-measure>>
3  (let* ((vals (iota 10 0.1 0.1))
4         (sines (map (lambda (i)
5                       (cdr (sine i)))
6                     vals)))
7    (zip vals sines))
```
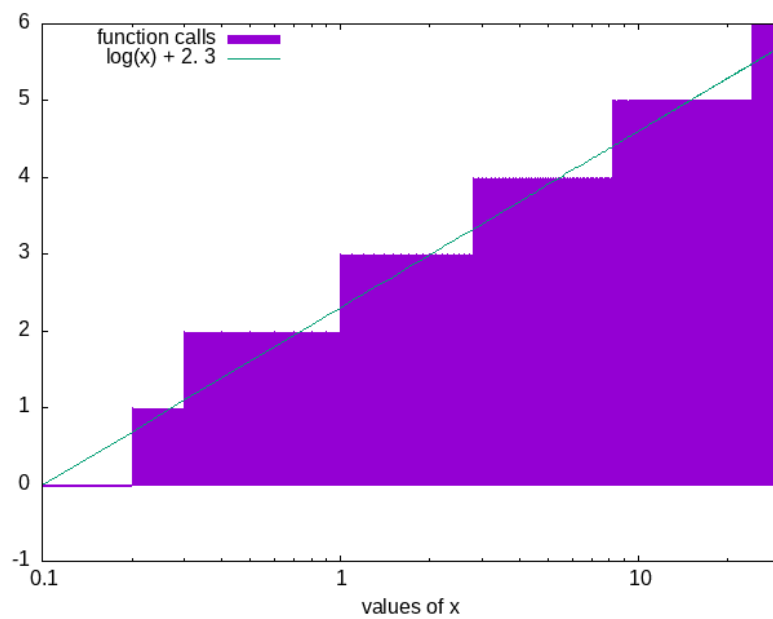
Example output:

35

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)))
```

```
4   (p (p (p (sine 0.45))))
5   (p (p (p (p (sine 0.15)))))
6   (p (p (p (p (p (sine 0.05))))))
7   (p (p (p (p (p 0.05)))))
8   (p (p (p (p 0.14950000000000002))))
9   (p (p (p 0.43513455050000005)))
10  (p (p 0.9758465331678772))
11  (p -0.7895631144708228)
12  -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:

```
1   ;; This execution takes too long for org-mode, so I'm doing it
2   ;; externally and importing the results
3   (use-srfis '(1))
4   (use-modules (ice-9 format))
5   (load "../../mattbench.scm")
6   <<1-15-deps>>
7   (let* ((vals (iota 300 0.1 0.1))
8          (times (map (lambda (i)
9                        (mattbench (lambda () (sine i)) 1000000))
10                     vals)))
11     (with-output-to-file "sine-bench.dat" (lambda ()
12       (map (lambda (x y)
13             (format #t "~s~/~s~%" x y))
14           vals times))))
```

```
1   reset # helps with various issues in execution
2   set xtics 0.5
3   set xlabel 'values of x'
4   set logscale x
5   set key top left
6   set style fill solid 1.00 border
7   #set style function fillsteps below
8
9   f(x) = (log(x) * a) + b
10  fit f(x) 'Ex15/sine-bench.dat' using 1:2 via a,b
11
12  plot 'Ex15/sine-bench.dat' using 1:2 with fillsteps title
    ↪  'time to execute', \
13      'Ex15/sine-bench.dat' using 1:(f($1)) with lines title
        ↪  sprintf('(log(x) * %.2f) + %.2f', a, b)
```
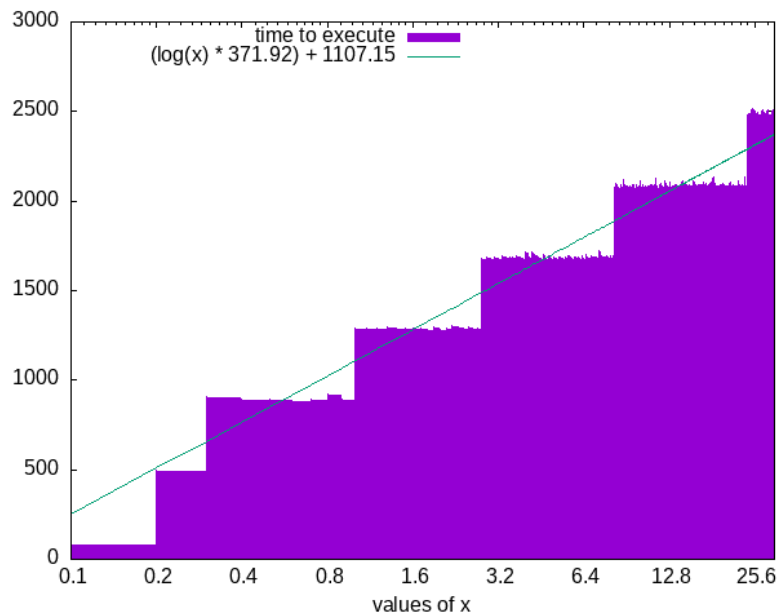
1. 1.2.4 Exponentiation Considering a few ways to compute the exponential of a given number.

```scheme
(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: Making `fast-expt` iterative

2.28.1 Text

38

```
1   (define (expt-rec b n)
2     (if (= n 0)
3         1
4         (* b (expt-rec b (- n 1)))))
5
6   (define (expt-iter b n)
7     (define (iter counter product)
8       (if (= counter 0)
9           product
10          (iter (- counter 1)
11                (* b product))))
12    (iter n 1))
13
14  (define (fast-expt b n)
15    (cond ((= n 0)
16           1)
17          ((even? n)
18           (square (fast-expt b (/ n 2))))
19          (else
20           (* 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:

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

```
1   <<square>>
2   (define (fast-expt-iter b n)
3     (define (iter b n a)
4       (format #t "~&|~s~/~/|~s~/~/|~s|~%" b n a)
5       (cond ((= n 1) (begin (format #t "~&|~s~/~/|~s~/~/|~s|~%" (* b a) 1)
     ↪  1)
6                             (* b a)))
```

39

```
 7              ((even? n) (iter (square b)
 8                               (/ n 2)
 9                               a))
10              (else (iter b (- n 1) (+ a 1)))))))
11      (format #t "|~a~/|~a~/|~a|~%" "base" "power" "variable")
12      (format #t "~&|--|--|--|~%")
13      (iter b n 1))
```

```
1   <<fast-expt-iter-fail1>>
2   <<try-these>>
3   (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 $a \to a \times b$

```
1   <<square>>
2   (define (fast-expt-iter b n)
3     (define (iter b n a)
4       (cond ((= n 1) (* b a))
5             ((even? n) (iter (square b)
6                              (/ n 2)
```

```
7                          a))
8             (else (iter b (- n 1) (* b a)))))))
9     (iter b n 1))
```

```
1   <<fast-expt-iter>>
2   <<try-these>>
3   (try-these (lambda(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: Logarithmic multiplication (recursive)

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:

```
1   (define (* a b)
2     (if (= b 0)
3         0
4         (+ 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

```
1   (define (double x)
2     (+ x x))
3   (define (halve x)
4     (/ x 2))
```

```
5   (define (fast-mult-rec a b)
6     (cond ((= b 0) 0)
7           ((even? b)
8            (double (fast-mult-rec a (halve b)))) ; This was kind of a
             ↪ stretch to think of.G
9            ;(fast-mult (double a) (halve b))) ⟸ My first instinct is
             ↪ iterative
10           (else (+ a (fast-mult-rec a (- b 1))))))))
```

Proof it works:

```
1   <<fast-mult-rec>>
2   <<try-these>>
3   (try-these (lambda(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: Logarithmic multiplication (iterative)

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:

```
1   (load "../mattbench.scm")
2   <<fast-mult-iter>>
3   <<fast-mult-rec>>
4   <<print-table>>
5   (print-table (list (list "fast-mult-rec" "fast-mult-iter")
6                      (list (mattbench (lambda() (fast-mult-rec 32
                       ↪ 32)) 10000000)
```

```
7                              (mattbench (lambda() (fast-mult 32 32))
                               ↪   10000000)))
8                    #: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

```
1   (define (double x)
2     (+ x x))
3   (define (halve x)
4     (/ x 2))
5   (define (fast-mult a b)
6     (define (iter a b c)
7       (cond ((= b 0) 0)
8             ((= b 1) (+ a c))
9             ((even? b)
10             (iter (double a) (halve b) c))
11            (else (iter a (- b 1) (+ a c)))))
12    (iter a b 0))
```

```
1   <<fast-mult-iter>>
2   <<try-these>>
3   (try-these (lambda(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

43

2.31 Exercise 1.19: Logarithmic fibonacci computing with T

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:

$$a < -a + b \text{ and } b < -a$$

Call this transformation T, and observe that applying T over and over again n times, starting with 1 and 0, produces the pair Fib($n+1$) and Fib(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)
  (lambda (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)))))
```

$$T_{pq}\colon \text{a,b} \mapsto \begin{cases} a \leftarrow \text{bq} + \text{aq} + \text{ap} \\ b \leftarrow \text{bp} + \text{aq} \end{cases}$$

```
<<T-func>>
<<try-these>>
(try-these (lambda (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))
```

```
 5                    (fib-rec (- n 2))))))
 6   (define (fib n)
 7     (fib-iter 1 0 0 1 n))
 8
 9   (define (fib-iter a b p q count)
10     (cond ((= count 0) b)
11           ((even? count)
12            (fib-iter a
13                      b
14                      (+ (* p p)
15                         (* q q))      ; compute p'
16                      (+ (* p q)
17                         (* q q)
18                         (* q p))      ; compute q'
19                      (/ count 2)))
20           (else (fib-iter (+ (* b q) (* a q) (* a p))
21                           (+ (* b p) (* a q))
22                           p
23                           q
24                           (- 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.

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

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 kth Fibonacci number.

2.33 Exercise 1.20: Inefficiency of normal-order evaluation

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 Substitution Model for Procedure Application. (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)
```

```
7   (gcd 6 (remainder 40 6))
8   eval remainder before call
9   call (gcd 6 4)
10  (if)
11  (gcd 2 (remainder 4 2))
12  eval remainder before call
13  call (gcd 2 0)
14  (if)
15  ;; ⇒ 2
```

```
1   ;; call gcd
2   (gcd 206 40)
3
4   ;; eval conditional
5   (if (= 40 0)
6       206
7       (gcd 40 (remainder 206 40)))
8
9   ;; recurse
10  (gcd 40 (remainder 206 40))
11
12  ; encounter conditional
13  (if (= (remainder 206 40) 0)
14      40
15      (gcd (remainder 206 40)
16           (remainder 40 (remainder 206 40))))
17
18  ; evaluate 1 remainder
19  (if (= 6 0)
20      40
21      (gcd (remainder 206 40)
22           (remainder 40 (remainder 206 40))))
23
24  ; recurse
25  (gcd (remainder 206 40)
26       (remainder 40 (remainder 206 40)))
27
28  ; encounter conditional
29  (if (= (remainder 40 (remainder 206 40)) 0)
30      (remainder 206 40)
31      (gcd (remainder 40 (remainder 206 40))
32           (remainder (remainder 206 40) (remainder 40 (remainder
            ↪  206 40)))))
33
34  ; eval 2 remainder
35  (if (= 4 0)
36      (remainder 206 40)
37      (gcd (remainder 40 (remainder 206 40))
```

```scheme
38              (remainder (remainder 206 40) (remainder 40 (remainder
     ↪    206 40)))))

39
40   ; recurse
41   (gcd (remainder 40 (remainder 206 40))
42        (remainder (remainder 206 40) (remainder 40 (remainder 206
     ↪    40))))

43
44   ; encounter conditional
45   (if (= (remainder (remainder 206 40) (remainder 40 (remainder 206
     ↪    40))) 0)
46        (remainder 40 (remainder 206 40))
47        (gcd (remainder (remainder 206 40) (remainder 40 (remainder
     ↪    206 40)))
48              (remainder (remainder 40 (remainder 206 40)) (remainder
                 ↪    (remainder 206 40) (remainder 40 (remainder 206
                 ↪    40))))))

49
50   ; eval 4 remainders
51   (if (= 2 0)
52        (remainder 40 (remainder 206 40))
53        (gcd (remainder (remainder 206 40) (remainder 40 (remainder
     ↪    206 40)))
54              (remainder (remainder 40 (remainder 206 40)) (remainder
                 ↪    (remainder 206 40) (remainder 40 (remainder 206
                 ↪    40))))))

55
56   ; recurse
57   (gcd (remainder (remainder 206 40) (remainder 40 (remainder 206
     ↪    40)))
58        (remainder (remainder 40 (remainder 206 40)) (remainder
         ↪    (remainder 206 40) (remainder 40 (remainder 206 40)))))

59
60   ; encounter conditional
61   (if (= (remainder (remainder 40 (remainder 206 40)) (remainder
     ↪    (remainder 206 40) (remainder 40 (remainder 206 40)))) 0)
62        (remainder (remainder 206 40) (remainder 40 (remainder 206
         ↪    40)))
63        (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)))))))

64
65   ; eval 7 remainders
66   (if (= 0 0)
67        (remainder (remainder 206 40) (remainder 40 (remainder 206
         ↪    40)))
```

49

```
68        (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)))))))))
69
70  ; eval 4 remainders
71  (remainder (remainder 206 40) (remainder 40 (remainder 206 40)))
72  ; ⇒ 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 nth 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

```
1  <<square>>
2  (define (smallest-divisor n)
3    (find-divisor n 2))
4
5  (define (find-divisor n test-divisor)
6    (cond ((> (square test-divisor) n)
```

```
7            n)
8           ((divides? test-divisor n)
9            test-divisor)
10          (else (find-divisor
11                n
12                (+ test-divisor 1)))))))
13
14  (define (divides? a b)
15    (= (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

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

$$199 \quad 1999 \quad 7$$

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.

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

```
1  <<prime-smallest-divisor>>
2  (define (timed-prime-test n)
3    (newline)
4    (display n) ;; Guile compatible \downarrow
5    (start-prime-test n (get-internal-run-time)))
6  (define (start-prime-test n start-time)
7    (if (prime? n)
8        (begin
9          (report-prime (- (get-internal-run-time)
10                       start-time))
```

```
11          n)
12        #f))
13  (define (report-prime elapsed-time)
14    (display " *** ")
15    (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.

```
1   <<timed-prime-test-txt>>
2   (define (search-for-primes minimum goal)
3     (define m (if (even? minimum)
4                   (+ minimum 1)
5                   (minimum)))
6     (search-for-primes-iter m '() goal))
7   (define (search-for-primes-iter n lst goal)
8     (if (= goal 0)
9         lst
10        (let ((x (timed-prime-test n)))
11          (if (not (equal? x #f))
12              (search-for-primes-iter (+ n 2) (cons x lst) (- goal
              ↪  1))
13              (search-for-primes-iter (+ n 2) lst goal)))))
```

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

```
1   1001
2   1003
3   1005
4   1007
5   1009 *** 1651
```

52

```
 6   1011
 7   1013 *** 1425
 8   1015
 9   1017
10   1019 *** 1375
```

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

```
 1   <<search-primes-basic>>
 2   (let ((lt1000-1 (search-for-primes 1000 3))
 3         (lt10000-1 (search-for-primes 10000 3))
 4         (lt100000-1 (search-for-primes 100000 3))
 5         (lt100000000-1 (search-for-primes 1000000 3)))
 6     (list
 7      (list "Primes > 1000" (reverse lt1000-1))
 8      (list "Primes > 10000" (reverse lt10000-1))
 9      (list "Primes > 100000" (reverse lt100000-1))
10      (list "Primes > 100000000" (reverse lt100000000-1))
11      ))
```

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.

```
 1   (define (mattbench2 f n)
 2     ;; Executes "f" for n times, and returns how long it took.
 3     ;; f is a lambda that takes no arguments, a.k.a. a "thunk"
 4
 5     ;; Returns a list with car(last execution results) and cadr(time
 ↪     taken divided by iterations n)
 6
 7     (define (time-getter) (get-internal-run-time))
 8     (define start-time (time-getter))
 9     (define (how-long) (- (time-getter) start-time))
10
11     (define (iter i)
12       (f)
13       (if (≤ i 0)
14           (f) ;; return the results of the last function call
15           (iter (- i 1))))
16
17     (list (iter n) ;; result of last call of f
```

53

```
18        (/ (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.

```
1    (define (search-for-primes minimum goal)
2      (define m (if (even? minimum)
3                    (+ minimum 1)
4                    (minimum)))
5      (search-for-primes-iter m '() goal))
6    (define (search-for-primes-iter n lst goal)
7      (if (= goal 0)
8          lst
9          (let ((x (prime? n)))
10           (if (not (equal? x #f))
11               (search-for-primes-iter (+ n 2) (cons n lst) (- goal
                 ↪   1))
12               (search-for-primes-iter (+ n 2) lst goal)))))
```

I can benchmark these functions like so:

```
1    <<mattbench2>>
2    <<prime-smallest-divisor>>
3    <<search-for-primes-untimed>>
4    <<print-table>>
5
6    (define benchmark-iterations 1000000)
7    (define (testit f)
8      (list (cadr (mattbench2 (lambda() (f 1000 3))
         ↪   benchmark-iterations))
9            (cadr (mattbench2 (lambda() (f 10000 3))
                ↪   benchmark-iterations))
10           (cadr (mattbench2 (lambda() (f 100000 3))
                ↪   benchmark-iterations))
11           (cadr (mattbench2 (lambda() (f 1000000 3))
                ↪   benchmark-iterations))))
12
13   (print-row
14    (testit search-for-primes))
```

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

5425.223086   20772.332491   53577.240193   121986.712395

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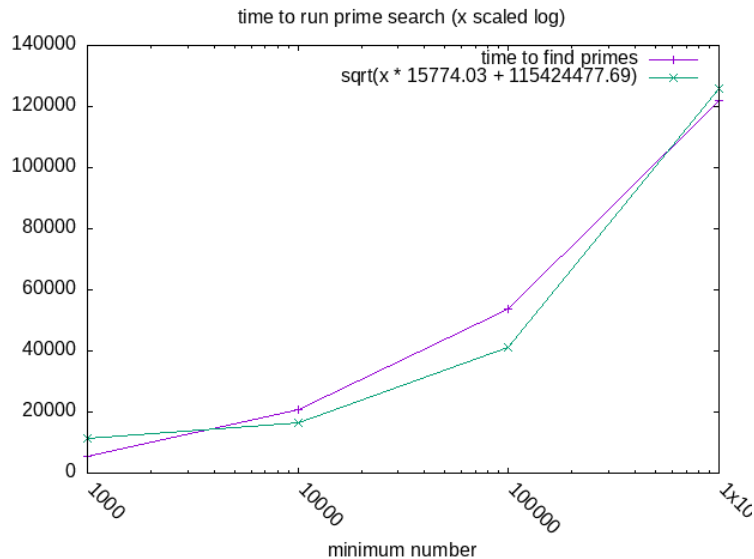
The plot for the square root function doesn't quite fit the real one and I'm not sure where the fault lies. I don't struggle to understand things like "this algorithm is slower than this other one," but when asked to find or prove the Θ notation I'm pretty clueless;

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 (`+` tes⌋ t-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)

55

```
1   <<square>>
2   (define (smallest-divisor n)
3     (find-divisor n 2))
4
5   (define (next n)
6     (if (= n 2)
7         3
8         (+ n 1)))
9
10  (define (find-divisor n test-divisor)
11    (cond ((> (square test-divisor) n)
12           n)
13          ((divides? test-divisor n)
14           test-divisor)
15          (else (find-divisor
16                  n
17                  (next test-divisor)))))
18
19  (define (divides? a b)
20    (= (remainder b a) 0))
```

```
1   <<mattbench2>>
2   <<find-divisor-faster>>
3   (define (prime? n)
4     (= n (smallest-divisor n)))
5   <<search-for-primes-untimed>>
6   <<print-table>>
7
8   (define benchmark-iterations 1000000)
9   (define (testit f)
10    (list (cadr (mattbench2 (lambda() (f 1000 3)) benchmark-iterations))
11          (cadr (mattbench2 (lambda() (f 10000 3)) benchmark-iterations))
12          (cadr (mattbench2 (lambda() (f 100000 3)) benchmark-iterations))
13          (cadr (mattbench2 (lambda() (f 1000000 3))
14          ↪ benchmark-iterations))))
15  (print-row
16   (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

time to run prime search (x scaled log)
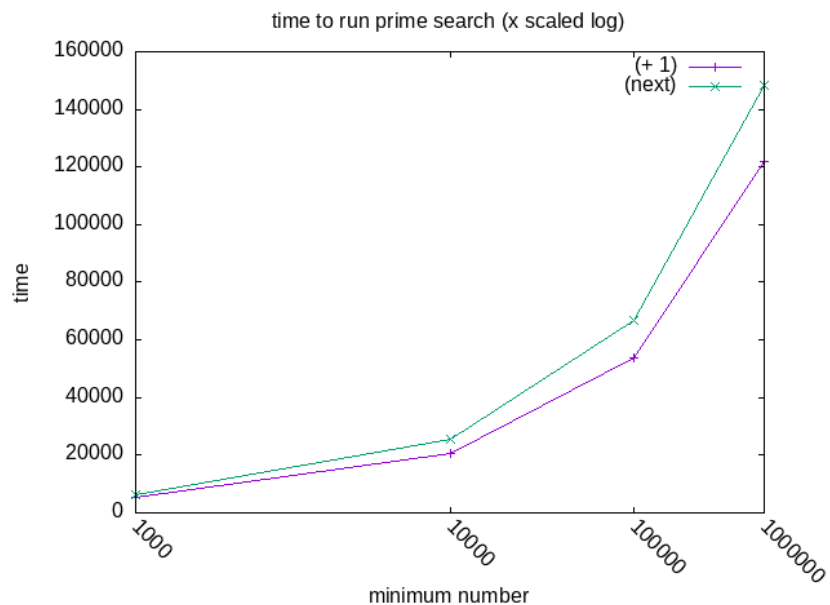
So it's *slower* than before. Why?

Oh, that's why.

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

2.37.3   Answer

Ok, let's try that again.

```
1   <<square>>
2   (define (smallest-divisor n)
3     (find-divisor n 2))
4
5   (define (next n)
6     (if (= n 2)
7         3
8         (+ n 2)))
9
10  (define (find-divisor n test-divisor)
11    (cond ((> (square test-divisor) n)
12           n)
13          ((divides? test-divisor n)
14           test-divisor)
```

```
15        (else (find-divisor
16               n
17               (next test-divisor)))))

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

```
1   <<mattbench2>>
2   <<find-divisor-faster-real>>
3   (define (prime? n)
4     (= n (smallest-divisor n)))
5   <<search-for-primes-untimed>>
6   <<print-table>>

8   (define benchmark-iterations 500000)
9   (define (testit f)
10    (list (cadr (mattbench2 (lambda() (f 1000 3)) benchmark-iterations))
11          (cadr (mattbench2 (lambda() (f 10000 3)) benchmark-iterations))
12          (cadr (mattbench2 (lambda() (f 100000 3)) benchmark-iterations))
13          (cadr (mattbench2 (lambda() (f 1000000 3))
              ↪  benchmark-iterations))))

15  (print-row
16   (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

58

time to run prime search (x scaled log)
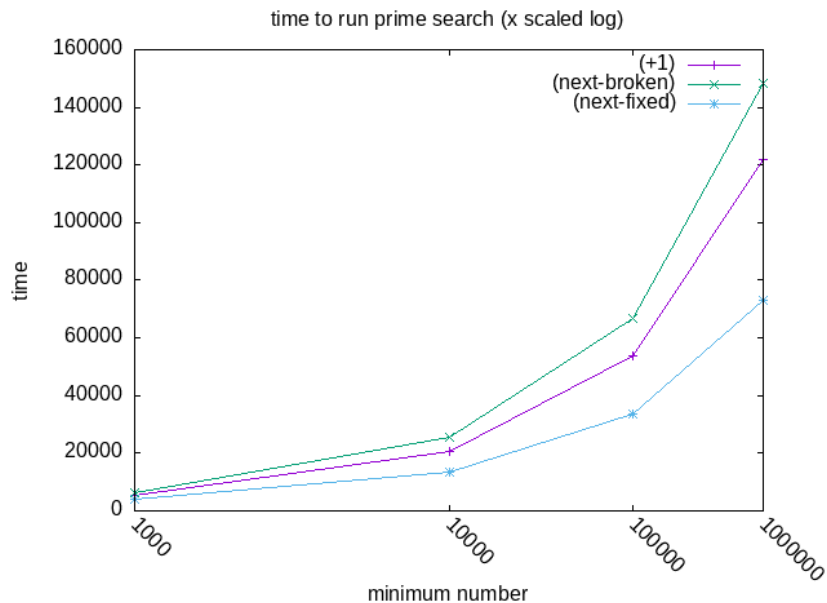
I had a lot of trouble getting this one to compile, I have to restart Emacs in order to get it to render.

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

```
1   <<square>>
2   (define (smallest-divisor n)
3     (if (divides? 2 n)                  ;; check for division by 2
4         2
5         (find-divisor n 3)))            ;; start find-divisor at 3
6
7   (define (find-divisor n test-divisor)
8     (cond ((> (square test-divisor) n)
9            n)
10          ((divides? test-divisor n)
11           test-divisor)
12          (else (find-divisor
13                 n
14                 (+ 2 test-divisor)))))) ;; just increase by 2
15
16  (define (divides? a b)
17    (= (remainder b a) 0))
```

```
1   <<mattbench2>>
2   <<find-divisor-faster-real2>>
3   (define (prime? n)
4     (= n (smallest-divisor n)))
5   <<search-for-primes-untimed>>
6   <<print-table>>
7
8   (define benchmark-iterations 500000)
9   (define (testit f)
10    (list (cadr (mattbench2 (lambda() (f 1000 3)) benchmark-iterations))
11          (cadr (mattbench2 (lambda() (f 10000 3)) benchmark-iterations))
12          (cadr (mattbench2 (lambda() (f 100000 3)) benchmark-iterations))
13          (cadr (mattbench2 (lambda() (f 1000000 3))
14           ↪   benchmark-iterations))))
15  (print-row
16   (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

time to run prime search (x scaled log)
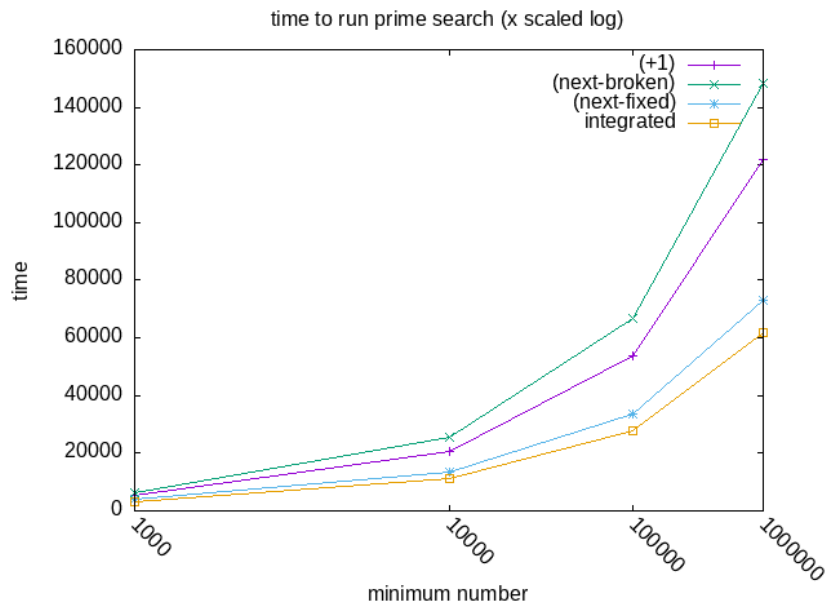
% speedup for broken (next)	81.93%
% speedup for real (next)	155.25%
% speedup for optimized	186.59%

2.38   Exercise 1.24

2.38.1   Text

```
1  <<square>>
2  (define (expmod base exp m)
3    (cond ((= exp 0) 1)
4          ((even? exp)
5           (remainder
6            (square (expmod base (/ exp 2) m))
7            m))
8          (else
9           (remainder
```

61

```
10            (* base (expmod base (- exp 1) m))
11            m))))
```

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

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

2.38.2   Question

Modify the `timed-prime-test` procedure of **??** 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

```
1  <<mattbench2>>
2  <<expmod>>
3  <<fermat-test>>
4  <<fast-prime>>
5  (define fermat-iterations 2)
6  (define (prime? n)
7    (fast-prime? n fermat-iterations))
8  <<search-for-primes-untimed>>
9  <<print-table>>
10
11 (define benchmark-iterations 500000)
12 (define (testit f)
13   (list (cadr (mattbench2 (lambda() (f 1000 3)) benchmark-iterations))
14         (cadr (mattbench2 (lambda() (f 10000 3)) benchmark-iterations))
15         (cadr (mattbench2 (lambda() (f 100000 3)) benchmark-iterations))
16         (cadr (mattbench2 (lambda() (f 1000000 3))
17           ↪  benchmark-iterations))))
18 (print-row
19  (testit search-for-primes))
```
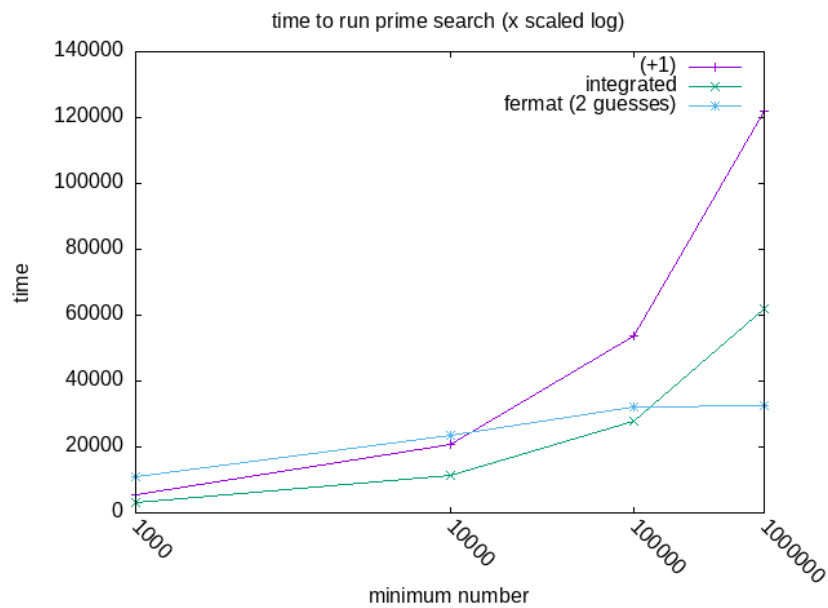
11175.799722   23518.62116   32150.745642   32679.766448

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

time to run prime search (x scaled log)

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

```
1   <<mattbench2>>
2   <<find-divisor-faster-real>>
3   (define (prime? n)
4     (= n (smallest-divisor n)))
5   <<search-for-primes-untimed>>
6   <<print-table>>
7
8   (define benchmark-iterations 100000)
9   (define (testit f)
10    (list (cadr (mattbench2 (lambda() (f 1000 3)) benchmark-iterations))
11          (cadr (mattbench2 (lambda() (f 10000 3)) benchmark-iterations))
12          (cadr (mattbench2 (lambda() (f 100000 3)) benchmark-iterations))
13          (cadr (mattbench2 (lambda() (f 1000000 3)) benchmark-iterations))
```

```
14        (cadr (mattbench2 (lambda() (f 10000000 3))
          ↪  benchmark-iterations))
15        (cadr (mattbench2 (lambda() (f 100000000 3))
          ↪  benchmark-iterations))
16        (cadr (mattbench2 (lambda() (f 1000000000 3))
          ↪  benchmark-iterations))
17        (cadr (mattbench2 (lambda() (f 10000000000 3))
          ↪  benchmark-iterations))
18        (cadr (mattbench2 (lambda() (f 100000000000 3))
          ↪  benchmark-iterations))
19        (cadr (mattbench2 (lambda() (f 1000000000000 3))
          ↪  benchmark-iterations))))

20
21  (print-row
22   (testit search-for-primes))
```

```
1   <<mattbench2>>
2   <<expmod>>
3   <<fermat-test>>
4   <<fast-prime>>
5   (define fermat-iterations 100)
6   (define (prime? n)
7     (fast-prime? n fermat-iterations))
8   <<search-for-primes-untimed>>
9   <<print-table>>

10
11  (define benchmark-iterations 100000)
12  (define (testit f)
13    (list (cadr (mattbench2 (lambda() (f 1000 3)) benchmark-iterations))
14          (cadr (mattbench2 (lambda() (f 10000 3)) benchmark-iterations))
15          (cadr (mattbench2 (lambda() (f 100000 3)) benchmark-iterations))
16          (cadr (mattbench2 (lambda() (f 1000000 3)) benchmark-iterations))
17          (cadr (mattbench2 (lambda() (f 10000000 3))
          ↪  benchmark-iterations))
18          (cadr (mattbench2 (lambda() (f 100000000 3))
          ↪  benchmark-iterations))
19          (cadr (mattbench2 (lambda() (f 1000000000 3))
          ↪  benchmark-iterations))
20          (cadr (mattbench2 (lambda() (f 10000000000 3))
          ↪  benchmark-iterations))
21          (cadr (mattbench2 (lambda() (f 100000000000 3))
          ↪  benchmark-iterations))
22          (cadr (mattbench2 (lambda() (f 1000000000000 3))
          ↪  benchmark-iterations))))

23
24  (print-row
25   (testit search-for-primes))
```
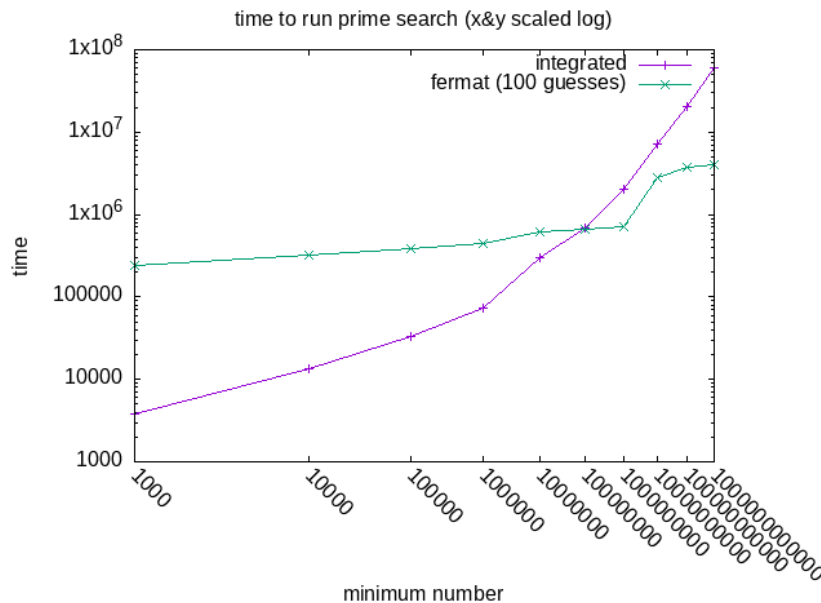
For the life of me I have no idea what that bump is. Maybe it needs more aggressive bignum processing there?

2.39 Exercise 1.25

2.39.1 Question

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

```
(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 1**⌋ **001**):

```
1   (expmod 455 1001 1001)
2   >  (even? 1001)
3   >  #f
4   >  (expmod 455 1000 1001)
5   >  >  (even? 1000)
6   >  >  #t
7   >  >  (expmod 455 500 1001)
8   >  >  >  (even? 500)
9   >  >  >  #t
10  ;; ...
11  >  >  >  x11 (expmod 455 2 1001)
12  >  >  >  x11 >  (even? 2)
13  >  >  >  x11 >  #t
14  >  >  >  x11 >  (expmod 455 1 1001)
15  >  >  >  x11 >  >  (even? 1)
16  >  >  >  x11 >  >  #f
17  >  >  >  x11 >  >  (expmod 455 0 1001)
18  >  >  >  x11 >  >  1
19  >  >  >  x11 >  455
20  >  >  >  x11 >  (square 455)
21  >  >  >  x11 >  207025
22  >  >  >  x11 819
23  ;; ...
24  >  >  >  (square 364)
25  >  >  >  132496
26  >  >  364
27  >  >  (square 364)
28  >  >  132496
29  >  364
30  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`:

```
1   (define (expmod base exp m)
2     (cond ((= exp 0) 1)
3           ((even? exp)
4            (remainder
5             (* (expmod base (/ exp 2) m)  ;;  <== hmm.
6                (expmod base (/ exp 2) m))
7             m))
8           (else
9            (remainder
10            (* base
11               (expmod base (- exp 1) m))
12            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

```
1   <<expmod>>
2   (define (car-test n)
3     (define (check a)
4       (= (remainder (expt a n) n)
5          (remainder (modulo a n) n)))
6     (every check
7            (cddr (iota n))))
```

```
1  <<car-test>>
2  (list (car-test 12) ; ⟸ false (not prime)
3        (car-test 1009);⟸ true  (real prime)
4        (car-test 561));⟸ true  (not prime,
5                        ;       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:

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

```
1  ;; fermat prime test but checks *every* value from 2 to n-1
2  (define (fermat-prime? n)
3    (define (iter a)
4      (if (= a n)
```

```
5          #f
6          (if (= (expmod a n n) a)
7              #t
8              (iter (+ 1 a)))))))
9    (iter 2))
```

```
1    (use-srfis '(1))
2    <<expmod>>
3    <<fermat-prime?>>
4    <<find-divisor-faster-real>>
5    (define (prime? n)
6      (= n (smallest-divisor n)))
7    (call-with-output-file "Data/carmichael-verification.txt" (lambda(port)
8        (write (filter
9                (lambda(x) (and (fermat-prime? x)
10                           (not (prime? x))))
11               (iota (- 1000000 1000) 1000))
12             port)))
```

This will be useful in various future functions:

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

```
1    (use-srfis '(1))
2    <<expmod>>
3    <<fermat-prime?>>
4    <<find-divisor-faster-real>>
5    (define (prime? n)
6      (= n (smallest-divisor n)))
7    <<get-lists-of-primes>>
8    (define prime-is-working
9      (and (and-map prime? list-of-primes)
10          (not (and-map prime? list-of-carmichaels))))
11   (format #t "(prime?) is working: ~a~%"
12          (if prime-is-working
13              "Yes"
14              "No"))
15   (define fermat-is-vulnerable
16     (and (and-map fermat-prime? list-of-primes)
17          (and-map fermat-prime? list-of-carmichaels)))
18   (format #t "(fermat-prime?) is vulnerable: ~a~%"
19          (if fermat-is-vulnerable
```

```
20          "Yes"
21          "No"))
```

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

2.42.3   Answer

```
1   <<square>>
2   (define (expmod-mr base exp m)
3     (cond ((= exp 0) 1)
4           ((even? exp)
5            (let ((sqr
6                   (square (expmod-mr base (/ exp 2) m))))
7              (if (= 1 (modulo sqr m))
8                  0
9                  (remainder sqr m))))
10          (else
11           (remainder
12            (* base (expmod-mr base (- exp 1) m))
13            m))))
```

```
1   (define (mr-test n)
2     (define (try-it a)
3       (let ((it (expmod-mr a n n)))
4         (or (= it a)
5             (= it 0))))
6     (try-it (+ 1 (random (- n 1)))))
```

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

```
1   <<expmod-mr>>
2   <<mr-test>>
3   <<mr-prime>>
4   (define mr-times 100)
5   <<get-lists-of-primes>>
6   (format #t
↪   "         mr detects primes: ~a~%mr false-positives Carmichaels: ~a~%"
7           (and-map (lambda(x)(mr-prime? x mr-times)) list-of-primes)
8          (and-map (lambda(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.

```
1   <<square>>
2   (define (expmod-mr base exp m)
3     (cond ((= exp 0) 1)
4           ((even? exp)
5            ;; Keep result and remainder seperate
6            (let* ((result (expmod-mr base (/ exp 2) m))
7                   (rem (remainder (square result) m)))
8              (if (and (not (= result 1))
9                       (not (= result (- m 1)))
10                      (= 1 rem))
11                 0 ;; non-trivial sqrt mod 1 is found
12                 rem)))
13          (else
14           (remainder
15            (* base (expmod-mr base (- exp 1) m))
16            m)))))
```

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

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

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

One more time.

```
1   (define (mr-test n)
2     (define (try-it a)
3       (= 1 (expmod-mr a (- n 1) n)))
4     (try-it (+ 1 (random (- n 1)))))
```

```
1   <<expmod-mr2>>
2   <<mr-test2>>
3   <<mr-prime>>
4   (define mr-times 100)
5   <<get-lists-of-primes>>
```

```
6   (format #t
    ↪  "        mr detects primes: ~a~%mr false-positives Carmichaels: ~a~%"
7          (and-map (lambda(x)(mr-prime? x mr-times)) list-of-primes)
8        (and-map (lambda(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:

```
1   (define (sum term a next b)
2     (if (> a b)
3         0
4         (+ (term a)
5            (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

```
1   (define (sum term a next b)
2     (if (> a b)
3         0
4         (+ (term a)
```

```
5              (sum term (next a) next b))))
```

```
1  (define (integral f a b dx)
2    (define (add-dx x)
3      (+ x dx))
4    (* (sum f (+ a (/ dx 2.0)) add-dx b)
5       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

```
1  (define (int-simp f a b n)
2    (define h
3      (/ (- b a)
4       n))
5    (define (gety k)
6      (f (+ a (* k h))))
7    (define (series-y sum k) ;; start with sum = y_0
8      (cond ((= k n) (+ sum (gety k)));; and k = 1
9            ((even? k) (series-y
10                        (+ sum (* 2 (gety k)))
11                        (+ 1 k)))
12            (else (series-y
13                   (+ sum (* 4 (gety k)))
14                   (+ 1 k)))))
15    (define sum-of-series (series-y (gety a) 1)) ;; (f a) = y_0
16    (* (/ h 3) sum-of-series))
```

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

```
1   <<mattbench2>>
2   <<print-table>>
3   (define (cube x)
4     (* x x x))
5   <<sum>>
6   <<integral>>
7   <<int-simp>>
8
9   (define iterations 100000) ;; benchmark iterations
10  (define (run-test1)
11    (integral cube 0.0 1.0 0.0008))
12  (define (run-test2)
13    (int-simp cube 0.0 1.0 1000.0))
14  (print-table (list (list "integral dx:0.0008" "int-simp i:1000")
15                     (list (run-test1) (run-test2))
16                     (list (cadr (mattbench2 run-test1 iterations))
17                           (cadr (mattbench2 run-test2 iterations))))
18               #: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

```
1   (define (sum-iter term a next b)
2     (define (iter a result)
3       (if (> a b)
4           result
5           (iter (next a) (+ result (term a)))))
6     (iter a 0))
```

Let's check the stats!

recursive	iterative
30051.080005	19568.685587

74

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

```
1  (define (product-iter term a next b)
2    (define (iter a result)
3      (if (> a b)
4          result
5          (iter (next a) (* result (term a)))))
6    (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.

```
1   <<product-iter>>
2   (define (identity x)
3     x)
4   (define (inc x)
5     (1+ x))
6
7   (define (factorial n)
8     (product-iter identity 1 inc n))
9
10  (display (factorial 7))
```

2.47.5 Question A.3

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

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

75

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 $\frac{1}{2}$.

```
1   <<product-iter>>
2   (define (pi-product n)
3     (define (div x)
4       (let ((x1 (- x 1))
5             (x2 (+ x 1)))
6         (* (/ x x1) (/ x x2))))
7     (* 2.0 (product-iter div 2 (lambda (z) (+ z 2)) n)))
8
9   (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

```
1   (define (product-rec term a next b)
2     (if (> a b)
3         1
4         (* (term a)
5            (product-rec term (next a) next b))))
```

```
1   <<mattbench2>>
2   <<print-table>>
3   <<product-iter>>
4   (define (pi-product n)
5     (define (div x)
6       (let ((x1 (- x 1))
7             (x2 (+ x 1)))
8         (* (/ x x1) (/ x x2))))
9     (* 2.0 (product-iter div 2 (lambda (z) (+ z 2)) n)))
10  <<product-rec>>
11  (define (pi-product-rec n)
12    (define (div x)
13      (let ((x1 (- x 1))
14            (x2 (+ x 1)))
15        (* (/ x x1) (/ x x2))))
16    (* 2.0 (product-rec div 2 (lambda (z) (+ z 2)) n)))
17
```

76

```
18  (define iterations 50000)
19  (print-table
20   (list (list "iterative" "recursive")
21         (list (cadr (mattbench2 (lambda()(pi-product 1000)) iterations))
22               (cadr (mattbench2 (lambda()(pi-product-rec 1000))
23                 ↪ 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:

```
1  (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.

```
1  (define (accumulate-iter combiner null-value term a next b)
2    (define (iter a result)
3      (if (> a b)
4          result
5          (iter (next a)
6                (combiner result (term a)))))
7    (iter a null-value))
```

```
1  <<accumulate-iter>>
2
3  ;; here you can see definitions in terms of accumulate
4  (define (sum term a next b)
5    (accumulate-iter + 0 term a next b))
6  (define (product term a next b)
```

```
7      (accumulate-iter * 1 term a next b))

8
9    (define (identity x)
10     x)
11   (define (inc x)
12     (1+ x))

13
14   ;; accumulate in action
15   (define (factorial n)
16     (accumulate-iter * 1 identity 1 inc n))

17
18   (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

```
1    (define (accumulate-rec combiner null-value term a next b)
2      (if (> a b)
3          null-value
4          (combiner (term a)
5             (accumulate-rec combiner null-value
6                             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

```
1    (define (filtered-accumulate-iter
2               predicate? combiner null-value
3               term a next b)
4      (define (iter a result)
5        (cond ((> a b) result)
```

```
 6              ((predicate? a)
 7               (iter (next a)
 8                     (combiner result (term a))))
 9              (else (iter (next a)
10                          result))))
11      (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)

```
 1    (load "mattcheck.scm")
 2    (define (square x)
 3      (* x x))
 4    <<filtered-accumulate-iter>>
 5    <<expmod-mr2>>
 6    <<mr-test2>>
 7    <<mr-prime>>
 8    (define mr-times 100)
 9    (define (prime? x)
10      (mr-prime? x mr-times))
11    (define (prime-sum a b)
12      (filtered-accumulate-iter prime? + 0
13                                square a 1+ b))
14
15    (mattcheck-equal "1 prime correct"
16                     (prime-sum 1008 1010)
17                     (square 1009)) ;; 1009
18    (mattcheck-equal "many primes correct"
19                     (prime-sum 1000 2001)
20                     (apply +
21                            (map square
22                                 (filter prime? (iota (- 2001 1000)
23                                                      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
   $\text{GCD}(i, n) = 1$.

79

```
1  (load "mattcheck.scm")
2  (define (square x)
3    (* x x))
4  (define (id x) x)
5  <<filtered-accumulate-iter>>
6  <<gcd>>
7  (define (relative-prime? x y)
8    (= 1 (gcd x y)))
9
10 (define (Ex_1-33B n)
11   (filtered-accumulate-iter
12    (lambda(i) (relative-prime? i n))
13    * 1 id
14    1 1+ (1- n)))
15
16 (define (alternate n)
17   (apply *
18         (filter (lambda(i) (relative-prime? i n))
19                 (iota (- n 1) 1))))
20
21 (mattcheck-equal "Ex_1-33B"
22                  (Ex_1-33B 100)
23                  (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

```
1  (define (f g) (g 2))
```

Then, we have

```
1   (f square)
2   ; 4
3   (f (lambda (z) (* z (+ z 1))))
4   ; 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.

```
1   ;; Will be evaluated like this:
2   ;;   (f f)
3   ;;   (f 2)
4   ;;   (2 2)
5   (define (f g) (g 2))
6   (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

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

```
1   (define tolerance 0.00001)
2
3   (define (fixed-point f first-guess)
4     (define (close-enough? v1 v2)
5       (< (abs (- v1 v2))
6          tolerance))
7     (define (try guess)
8       (let ((next (f guess)))
9         (if (close-enough? guess next)
10             next
11             (try next))))
12    (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 `fixe⌋ d-point` procedure.

2.53.3   Answer

```
1   <<close-enough>>
2   <<fixed-point-txt>>
3   (define golden-ratio
4     (fixed-point (lambda(x)(+ 1 (/ 1 x)))
5                  1.0))
6
7   (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.

```scheme
(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 (lambda(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 (lambda(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

distance from true golden ratio
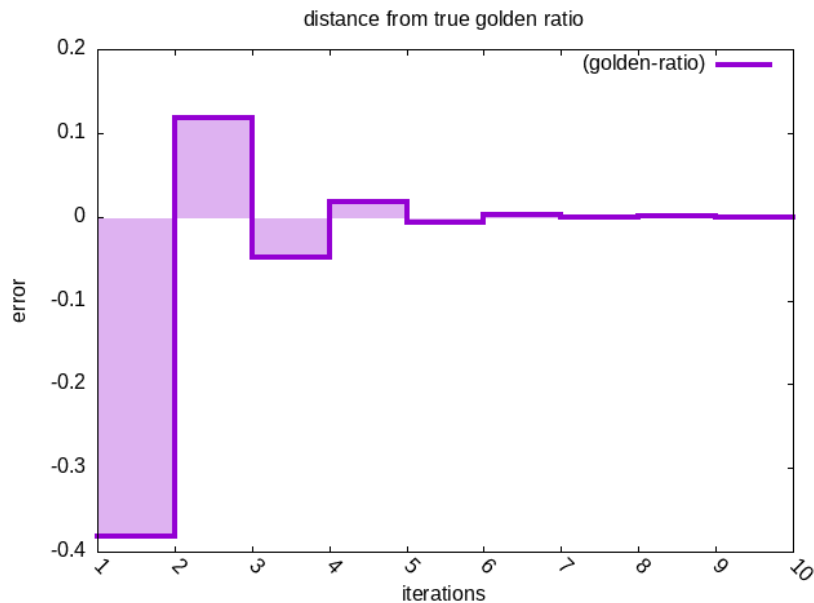
k must be at least 10 to get precision of 4 decimal places.

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

```
1  (define (cont-frac-rec n d k)
2    (define (rec i)
3      (if (= i k)
4          (/ (n i) (d i))
5          (/ (n i) (+ (d i) (rec (1+ i))))))
6
7    (rec 1))
```

```
1  <<cont-frac>>
2  <<cont-frac-rec>>
3  (define (golden-ratio k)
4    (cont-frac (lambda(i) 1.0)(lambda(i)1.0) k))
5  (define (golden-ratio-rec k)
6    (cont-frac-rec (lambda(i) 1.0)(lambda(i)1.0) k))
7
```

```
 8  (load "mattcheck.scm")
 9  (mattcheck-equal "cont-frac iter and recursive equivalence"
10              (golden-ratio-rec 15)
11              (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-fra‿c procedure from Exercise 1.37 to approximate e, based on Euler's expansion.

2.56.2 Answer

```
 1  <<cont-frac>>
 2  (define (euler k)
 3    (+ 2
 4       (cont-frac (lambda(i) 1.0)
 5                  (lambda(i) (let ((j (1+ i)))
 6                              (if (= 0 (modulo j 3))
 7                                  (* 2 (/ j 3))
 8                                  1)))
 9                  k)))
10
11  (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

```
1   <<cont-frac>>
2   (define (tan-cf x k)
3     (cont-frac (lambda(i) (if (= i 1)
4                             x
5                             (* x x -1.0)))
6               (lambda(i) (if (= i 1)
7                             1.0
8                             (- (* i 2.0) 1.0)))
9               k))
10
11  (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

```
1   (define (average-damp f)
2     (lambda (x) (average x (f x))))
```

```
1   (define dx 0.00001)
```

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

```
1  (define (newton-transform g)
2    (lambda (x) (- x (/ (g x) ((deriv g) x)))))
3  (define (newtons-method g guess)
4    (fixed-point (newton-transform g) guess))
```

```
1  <<average>>
2  <<average-damp>>
3  <<dx>>
4  <<deriv>>
5  <<newtons-method>>
```

2.59.2   Question

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

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

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

2.59.3   Answer

```
1  (define (cubic a b c)
2    (lambda (x)
3      (+ (expt x 3)
4         (* a (expt x 2))
5         (* b x)
6         c)))
```

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

```
1  <<fixed-point-txt>>
2  <<newtons-method-txt>>
3  <<cubic>>
4  <<cubic-zero>>
5
6  (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

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

2.60.2 Answer

```
1   (define (double f)
2     (lambda (x)
3       (f (f x))))
```

```
1   (define inc 1+)
2   <<double>>
3   <<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

```
1   (define (compose f g)
2     (lambda(x)
3       (f (g x))))
```

```
1   <<compose>>
2   <<square>>
3   (define inc 1+)
4   ((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 nth 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 nth repeated application of f is the function $x \mapsto x + n$. If f is the operation of squaring a number, then the nth 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 nth 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

```
1   <<average-varargs>>
2   (define (smooth f)
3     (lambda(x)
4       (average (f (- x dx))
5               (f x)
6               (f (+ x dx)))))
7   (define (smooth-n f n)
8     ((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:

```
1   <<fixed-point-txt>>
2   <<repeated>>
3   <<average-damp>>
4   (define (sqrt n)
5     (fixed-point
6       (average-damp
7         (lambda (y)
8           (/ x y)))
9       1.0))
10  (define (nth-root x n)
11    (fixed-point
12      ((repeated average-damp (ceiling (sqrt n)))
```

```
13        (lambda (y)
14          (/ x (expt y (- n 1)))))))
15      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.

```
1   ;; Version of "repeated" that can handle being asked to repeat zero
    ↪  times.
2   <<compose>>
3   <<identity>>
4   (define (repeated f n)
5     (define (rec m)
6     (if (= n 1)
7         f
8         (repeated (compose f f)
9                   (- n 1))))
10    (if (= n 0)
11        identity
12        (rec n)))
```

```
1   ;; version of "fixed-point" that will give up after a certain number of
    ↪  guesses.
2   (define (limited-fixed-point f first-guess)
3     (define limit 5000)
4     (define tolerance 0.00000001)
5     (define (close-enough? v1 v2)
6       (< (abs (- v1 v2))
7          tolerance))
8     (define (try guess tries)
9       (if (= tries limit)
10          "LIMIT REACHED"
11          (let ((next (f guess)))
12            (if (close-enough? guess next)
13                next
14                (try next (+ 1 tries))))))
15      (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.

```
1   <<fixed-point-txt>>
2   <<limited-fixed-point>>
3   <<repeated>>
4   <<average-damp>>
```
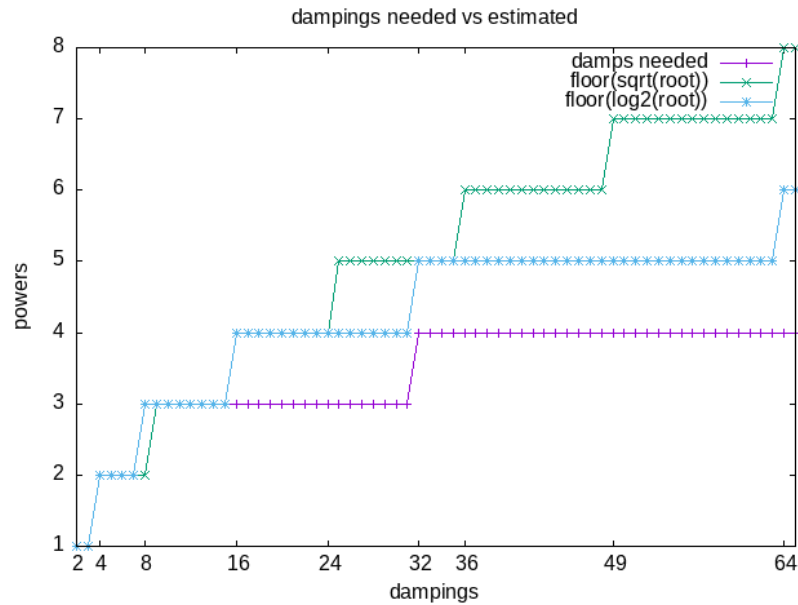
```
5    <<average>>
6    <<print-table>>
7    (define (sqrt x)
8      (fixed-point
9       (average-damp
10        (lambda (y) (/ x y)))
11       1.0))
12   (define (nth-tester base n-max)
13     (define (iter ll)
14       (let ((n (+ 2 (length ll))))
15         (define (try damps)
16           (let ((x (limited-fixed-point
17                     ((repeated average-damp damps)
18                      (lambda (y)
19                        (/ base (expt y (- n 1)))))
20                     1.1)))
21             (if (string? x)
22                 (try (1+ damps))
23                 (list base n x damps))))
24         (if (> n n-max)
25             ll
26             (iter (cons (try 1) ll)))))
27
28     (iter '()))
29   (let* ((t (reverse (nth-tester 3 65))))
30     (cons '("root" "result" "damps needed" "floor(sqrt(root))"
     ↪ "floor(log2(root))")
31          (map (lambda(x)
32                 (append x
33                        (list (floor (sqrt (car x)))
34                              (floor (/ (log (car x))(log 2))))))
35              (map cdr t))))
```

93

dampings needed vs estimated

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. (TODO)

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-imp`⌋
`rove`.

2.65.2   Answer

```
1  (define (iterative-improve good-enough? improve)
2    (define (iter guess)
3      (let ((next (improve guess)))
4        (if (good-enough? guess next)
5            next
6            (iter next))))
7    iter)
```

```
1  <<iterative-improve>>
2  (define tolerance 0.00001)
3
4  (define (fixed-point-improve f first-guess)
5    (define (close-enough? v1 v2)
6      (< (abs (- v1 v2))
7         tolerance))
8    ((iterative-improve
9      close-enough?
10     f)
11    first-guess))
```

```
1  <<square>>
2  <<average>>
3  <<iterative-improve>>
4  (define (improve guess x)
5    (average guess (/ x guess)))
6  (define (good-enough? next x)
7    ;;   (= guess next)) ;; no longer works with this method
8    (< (abs (- (square next) x)) 0.001))
9  (define (sqrt-improve x)
10   ((iterative-improve
11     (lambda(guess next)(good-enough? next x))
12     (lambda(guess) (improve guess x)))
13    1.0))
```

```
1  (load "mattcheck2.scm")
2  <<fixed-point-txt>>
3  <<fixed-point-improve>>
4  (mattcheck "fixed-point-improve still working"
5             (fixed-point (lambda(x)(+ 1 (/ 1 x))) 1.1)
6             (fixed-point-improve (lambda(x)(+ 1 (/ 1 x))) 1.1))
7  <<sqrt>>
8  <<sqrt-improve>>
9  (mattcheck "sqrt-improve still working"
10            (sqrt 5)
```

95

```
11                    (sqrt-improve 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   Text

```
1   (define (add-rat x y)
2     (make-rat (+ (* (numer x) (denom y))
3                  (* (numer y) (denom x)))
4               (* (denom x) (denom y))))
5
6   (define (sub-rat x y)
7     (make-rat (- (* (numer x) (denom y))
8                  (* (numer y) (denom x)))
9               (* (denom x) (denom y))))
10
11  (define (mul-rat x y)
12    (make-rat (* (numer x) (numer y))
13              (* (denom x) (denom y))))
14
15  (define (div-rat x y)
16    (make-rat (* (numer x) (denom y))
```

```
17              (* (denom x) (numer y))))

18

19   (define (equal-rat? x y)
20     (= (* (numer x) (denom y))
21        (* (numer y) (denom x))))
```

```
1    (define (make-rat n d) (cons n d))
2    (define (numer x) (car x))
3    (define (denom x) (cdr x))
```

```
1    (define (print-rat x)
2      (newline)
3      (display (numer x))
4      (display "/")
5      (display (denom x)))
```

```
1    (define one-half (make-rat 1 2))
2    (define one-third (make-rat 1 3))
3    (print-rat one-half)
4    (print-rat
5     (mul-rat one-half one-third))
```

```
1/2
1/6
```

3.2.2   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 denomi-
nator are positive, and if the rational number is negative, only the
numerator is negative.

3.2.3   Answer

```
1    <<abs>>
2    (define (make-rat n d)
3      (cond ((not (or (< n 0)
4                      (< d 0)))
5             (cons n d))
6            ((and (< n 0)
7                  (< d 0))
8             (cons (- n) (- d)))
9            (else
10            (cons (- (abs n)) (abs d)))))
```

```
11  (define (numer x) (car x))
12  (define (denom x) (cdr x))
13
14  ;; Bonus: an attempt to optimize
15  (define (make-rat-opt n d)
16    (let ((nn (< n 0))
17          (dn (< d 0)))
18      (cond ((not (or nn dn))
19             (cons n d))
20            ((and nn dn)
21             (cons (- n) (- d)))
22            (else
23             (cons (- (abs n)) (abs d))))))
```

```
1   <<make-rat>>
2   <<print-rat-txt>>
3   <<rat-ops-txt>>
4   (load "mattcheck2.scm")
5   (mattcheck "make-rat double negative"
6              (cons 1 2)
7              (make-rat -1 -2))
8   (mattcheck "make-rat numerator negative"
9              (cons -1 2)
10             (make-rat -1 2))
11  (mattcheck "make-rat denominator negative"
12             (cons -1 2)
13             (make-rat 1 -2))
14  (mattcheck "make-rat-opt double negative"
15             (cons 1 2)
16             (make-rat-opt -1 -2))
17  (mattcheck "make-rat-opt numerator negative"
18             (cons -1 2)
19             (make-rat-opt -1 2))
20  (mattcheck "make-rat-opt denominator negative"
21             (cons -1 2)
22             (make-rat-opt 1 -2))
```

```
SUCCEED at make-rat double negative
SUCCEED at make-rat numerator negative
SUCCEED at make-rat denominator negative
SUCCEED at make-rat-opt double negative
SUCCEED at make-rat-opt numerator negative
SUCCEED at make-rat-opt denominator negative
```

My "optimized" version shows no benefit at all:

```
unoptimized make-rat: ((1 . 2) 231.74267794)
optimized make-rat: ((1 . 2) 233.99087033)
```

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:

```
1  (define (print-point p)
2    (newline)
3    (display "(")
4    (display (x-point p))
5    (display ",")
6    (display (y-point p))
7    (display ")"))
```

3.3.2   Answer

```
1   <<average>>
2   (define (make-point x y)
3     (cons x y))
4   (define (x-point p)
5     (car p))
6   (define (y-point p)
7     (cdr p))
8   (define (make-segment start end)
9     (cons start end))
10  (define (start-segment seg)
11    (car seg))
12  (define (end-segment seg)
13    (cdr seg))
14  (define (midpoint-segment seg)
15    (make-point (average (x-point (start-segment seg))
16                         (x-point (end-segment seg)))
17               (average (y-point (start-segment seg))
18                         (y-point (end-segment seg)))))
19  (define (midpoint-segment-opt seg)
20    (let ((ax (x-point (start-segment seg)))
```

```scheme
21          (bx (x-point (end-segment seg)))
22          (ay (y-point (start-segment seg)))
23          (by (y-point (end-segment seg))))
24    (make-point (average ax
25                         bx)
26            (average ay
27                         by))))
```

```scheme
1   <<make-point>>
2   (load "mattcheck2.scm")
3   (mattcheck "make-point"
4             (list 1 2)
5             (let ((p (make-point 1 2)))
6               (list (x-point p)
7                     (y-point p))))
8   (let* ((p1 (make-point 1 2))
9          (p2 (make-point -1 -2))
10         (s (make-segment p1 p2)))
11    (mattcheck "make-segment"
12             (list p1 p2)
13             (list (start-segment s)
14                   (end-segment s)))
15    (mattcheck "midpoint-segment"
16              (make-point 0 0)
17              (midpoint-segment s))
18    (mattcheck "midpoint-segment-opt"
19              (make-point 0 0)
20              (midpoint-segment-opt s)))
```

```
SUCCEED at make-point
SUCCEED at make-segment
SUCCEED at midpoint-segment
SUCCEED at midpoint-segment-opt
```

And once again my bikeshedding is revealed:

```
unoptimized make-rat: ((0.0 . 0.0) 326.94653558)
optimized make-rat: ((0.0 . 0.0) 331.83410742)
```

3.4   Exercise 2.3

3.4.1   Question

Implement a representation for rectangles in a plane. (Hint: You
may want to make use of Exercise 2.2.) In terms of your constructors
and selectors, create procedures that compute the perimeter and the
area of a given rectangle. Now implement a different representation
for rectangles. Can you design your system with suitable abstraction

barriers, so that the same perimeter and area procedures will work
using either representation?

3.4.2 Answer 1

I don't really like the "wishful thinking" process the book advocates but since
this question specifically regards abstraction, I'll start by writing the two re-
quested procedures first.

```
(define (rect-area R)
  (* (rect-height R)
     (rect-width R)))

(define (rect-peri R)
  (* 2
     (+ (rect-height R)
        (rect-width R))))
```

So my "wishlist" is just for (rect-area R) and (rect-width R).

So, my first implementation of a rectangle will be of a list of 3 points ABC,
with the fourth point D being constructed from the others. I haven't done
geometry lessons in a while but logically I can deduce that D is as far from A as
B is from C, and as far from C as A is from B. by experimentation I've figured
out that $D = A + (C - B) = C + (A - B)$.

```
;;    AB = width
;;(0,1) (1,1)
;;  A-----B
;;  |     | BC = height
;;  D-----C
;;(0,0) (1,0)
;; could be rotated any direction
<<square>>
<<make-point>>
(define (make-rect a b c)
  (cons (cons a b) c))
(define (rect-a R)
  (caar R))
(define (rect-b R)
  (cdar R))
(define (rect-c R)
  (cdr R))
;(define (rect-d R)
;  (make-point (x-point (rect-a R))
;              (y-point (rect-c R))))
;; Wait, this won't work if the rectangle is angled.

(define (sub-points a b)
```

101

```
24    (make-point (- (x-point a)
25                   (x-point b))
26                (- (y-point a)
27                   (y-point b)))))

29  (define (add-points a b)
30    (make-point (+ (x-point a)
31                   (x-point b))
32                (+ (y-point a)
33                   (y-point b)))))

35  (define (rect-d R)
36    (let ((a (rect-a R))
37          (b (rect-b R))
38          (c (rect-c R)))
39      (add-points a
40                  (sub-points c b))))
41  (define (rect-d-alt R) ; should be mathematically identical.
42    (let ((a (rect-a R))
43          (b (rect-b R))
44          (c (rect-c R)))
45      (add-points c
46                  (sub-points a b))))

48  ;; this is incorrect
49  ;(define (length-points a b)
50  ;   (let ((diffP (sub-points a b)))
51  ;     (+ (abs (x-point diffP))
52  ;        (abs (y-point diffP)))))
53  (define (length-points a b)
54    (let ((ax (x-point a))
55          (ay (y-point a))
56          (bx (x-point b))
57          (by (y-point b)))
58      (sqrt (+ (square (- ax bx))
59               (square (- ay by))))))

61  (define (rect-height R)
62    (abs (length-points (rect-b R)
63                        (rect-c R))))
64  (define (rect-width R)
65    (abs (length-points (rect-b R)
66                        (rect-a R))))

68  (define (length-segment seg)
69    (abs (length-points (start-segment seg)
70                        (end-segment seg))))
```

```scheme
(load "mattcheck2.scm")

<<rect-4pt>>
<<rect-area-peri>>

(let* ((a (make-point 13 14))
       (b (make-point 14 14))
       (c (make-point 14 13))
       (d (make-point 13 13))
       (ABC (make-rect a b c))
       (CDA (make-rect c d a))
       (w (make-point -2.0 -2.0))
       (x (make-point -0.5 -0.5))
       (y (make-point -1.5 0.5))
       (z (make-point -3.0 -1.0))
       (WXY (make-rect w x y)))
  (mattcheck "make-rect"
             ABC
             (cons (cons a b) c))
  (mattcheck "rect-d and rect-d-alt (ABCD)"
             (rect-d ABC)
             (rect-d-alt ABC)
             d)
  (mattcheck "rect-d and rect-d-alt (CDAB)"
             (rect-d CDA)
             (rect-d-alt CDA)
             b)
  (mattcheck "rect-d and rect-d-alt (WXYZ)"
             (rect-d WXY)
             (rect-d-alt WXY)
             z)
  (mattcheck "rect-d and rect-d-alt (XYZW)"
             (rect-d (make-rect x y z))
             w)
  (mattcheck "rect-height ABC"
             (rect-height ABC)
             1)
  (mattcheck "rect-width ABC"
             (rect-width ABC)
             1)
  (mattcheck "rect-height WXY"
             (rect-height WXY)
             1.4142135623730951)
  (mattcheck "rect-width WXY"
             (rect-width WXY)
             2.1213203435596424)
  (mattcheck "rect-area ABCD"
             (rect-area ABC)
```

103

```
49              (rect-area CDA)
50              1)
51    (mattcheck "rect-area WXYZ"
52              (rect-area WXY)
53              3.0)
54    (mattcheck "rect-peri ABCD"
55              (rect-peri ABC)
56              4)
57    (mattcheck "rect-peri WXYZ"
58              (rect-peri WXY)
59              7.0710678118654755))
```

```
SUCCEED at make-rect
SUCCEED at rect-d and rect-d-alt (ABCD)
SUCCEED at rect-d and rect-d-alt (CDAB)
SUCCEED at rect-d and rect-d-alt (WXYZ)
SUCCEED at rect-d and rect-d-alt (XYZW)
SUCCEED at rect-height ABC
SUCCEED at rect-width ABC
SUCCEED at rect-height WXY
SUCCEED at rect-width WXY
SUCCEED at rect-area ABCD
SUCCEED at rect-area WXYZ
SUCCEED at rect-peri ABCD
SUCCEED at rect-peri WXYZ
```

3.4.3   Answer 2

My second implementation will be of a rectangle as an origin, height, width, and angle. Basically, height and width are two vectors originating from origin, with width going straight right and height offset 90 deg from width. Angle is added during conversion from Polar to Cartesian coordinates. In relation to my 1st implementation, point D is where the origin is.

```
1   <<make-point>>
2   ;; origin is a (make-point), hwa are floats
3   (define (make-rect origin height width angle)
4     (cons (cons origin height)
5           (cons width angle)))
6
7   (define (rect-origin R)
8     (caar R))
9   (define rect-d rect-origin)
10  (define (rect-height R)
11    (cdar R))
12  (define (rect-width R)
13    (cadr R))
```

```
14   (define (rect-angle R)
15     (cddr R))
16
17   ;; I underestimated how much math this would take.
18   (define (add-points a b)
19     (make-point (+ (x-point a)
20                    (x-point b))
21                 (+ (y-point a)
22                    (y-point b))))
23
24   (define pi (* 4 (atan 1.0)))
25   (define (radian deg)
26     (* deg (/ pi 180.0)))
27   (define (vector-to-xy distance angle)
28         ;; rect-c: (cos(Theta),sin(Theta)) * width
29         (make-point (* (cos (radian angle))
30                        distance)
31                     (* (sin (radian angle))
32                        distance)))
33         ;; could also be rotated by 90 degrees just by using
34         ;;   (-sin(Theta),cos(Theta)) * height
35   (define (rect-c R)
36     (add-points
37      (rect-origin R)
38      (vector-to-xy (rect-width R) (rect-angle R))))
39   (define (rect-a R)
40     (add-points
41      (rect-origin R)
42      (vector-to-xy (rect-height R)
43                    (+ 90 (rect-angle R)))))
44   (define (rect-b R)
45     (add-points
46      (rect-origin R)
47      (add-points
48       (vector-to-xy (rect-width R) (rect-angle R))
49       (vector-to-xy (rect-height R)
50                     (+ 90 (rect-angle R))))))
```

```
1    (load "mattcheck2.scm")
2
3    <<rect-ohwa>>
4    <<rect-area-peri>>
5
6    (let* ((a (make-point 13.0 14.0))
7           (b (make-point 14.0 14.0))
8           (c (make-point 14.0 13.0))
9           (d (make-point 13.0 13.0))
10          (ABC (make-rect d 1 1 0))
```

```
11          (CDA (make-rect b 1 1 180))
12          (w (make-point -2.0 -2.0))
13          (x (make-point -2.5 1.5))
14          (y (make-point -1.5 0.5))
15          (z (make-point -3.0 -1.0))
16          (wxy-height 1.4142135623730951)
17          (wxy-width 2.1213203435596424)
18          (WXY (make-rect z wxy-height wxy-width 45)))
19     (mattcheck "make-rect"
20                ABC
21                (cons (cons d 1) (cons 1 0)))
22     (mattcheck "rect-b (ABCD)"
23                (rect-b ABC)
24                b)
25     (mattcheck "rect-b (CDAB)"
26                (rect-b CDA)
27                d)
28     (mattcheck "rect-b (WXYZ)"
29                (rect-b WXY)
30                x)
31     (mattcheck "rect-height"
32                (rect-height WXY)
33                wxy-height)
34     (mattcheck "rect-width"
35                (rect-width WXY)
36                wxy-width)
37     (mattcheck "rect-area ABCD"
38                (rect-area ABC)
39                (rect-area CDA)
40                1)
41     (mattcheck "rect-area WXYZ"
42                (rect-area WXY)
43                3.0)
44     (mattcheck "rect-peri ABCD"
45                (rect-peri ABC)
46                4)
47     (mattcheck "rect-peri WXYZ"
48                (rect-peri WXY)
49                7.0710678118654755))
```

```
SUCCEED at make-rect
SUCCEED at rect-b (ABCD)
SUCCEED at rect-b (CDAB)
SUCCEED at rect-b (WXYZ)
SUCCEED at rect-height
SUCCEED at rect-width
SUCCEED at rect-area ABCD
SUCCEED at rect-area WXYZ
SUCCEED at rect-peri ABCD
```

106

3.5   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.

3.6   Exercise 2.4

3.6.1   Question

Here is an alternative procedural representation of pairs. For this representation, verify that (`car` (`cons` x y)) yields x for any objects x and y.

```
(define (cons x y)
  (lambda (m) (m x y)))
(define (car z)
  (z (lambda (p q) p)))
```

What is the corresponding definition of `cdr`? (Hint: To verify that this works, make use of the substitution model of 1.1.5.)

3.6.2   Answer

First, let's explain with the substitution model.

```
(cons 0 1)
(lambda (m) (m 0 1))

(car (lambda (m) (m 0 1)))
((lambda (m) (m 0 1)) (lambda (p q) p))
(lambda (0 1) 0)
0
(cdr (lambda (m) (m 0 1)))
((lambda (m) (m 0 1)) (lambda (p q) q))
(lambda (0 1) 1)
1
```

Now for implementation.

```
1   (load "mattcheck2.scm")
2   <<alt-pairs-txt>>
3   (define (cdr z)
4     (z (lambda (p q) q)))
5
6   (let ((pair (cons 0 1)))
7     (mattcheck "car"
8                (car pair)
9                0)
10    (mattcheck "cdr"
11               (cdr pair)
12               1))
```

```
| (0 . 0) | (0 . 1) | (0 . 2) | (0 . 3) | (0 . 4) | (0 . 5) | (0 . 6) |
| (1 . 0) | (1 . 1) | (1 . 2) | (1 . 3) | (1 . 4) | (1 . 5) | (1 . 6) |
| (2 . 0) | (2 . 1) | (2 . 2) | (2 . 3) | (2 . 4) | (2 . 5) | (2 . 6) |
| (3 . 0) | (3 . 1) | (3 . 2) | (3 . 3) | (3 . 4) | (3 . 5) | (3 . 6) |
| (4 . 0) | (4 . 1) | (4 . 2) | (4 . 3) | (4 . 4) | (4 . 5) | (4 . 6) |
| (5 . 0) | (5 . 1) | (5 . 2) | (5 . 3) | (5 . 4) | (5 . 5) | (5 . 6) |
| (6 . 0) | (6 . 1) | (6 . 2) | (6 . 3) | (6 . 4) | (6 . 5) | (6 . 6) |
```

3.7   Exercise 2.5   optional

3.7.1   Question

Show that we can represent pairs of nonnegative integers using only
numbers and arithmetic operations if we represent the pair a and
b as the integer that is the product $2^a 3^b$. Give the corresponding
definitions of the procedures cons, car, and cdr.

3.7.2   Answer

This one really blew my mind inside-out when I first did it. Basically, because
the two numbers are coprime, you can factor out the unwanted number and be
left with the desired one.

Where x is the scrambled number, p is the base we want to remove, q
is the base we want to retrieve from and y is the value exponentiating
p, the original number is retrieved by dividing x by p for y number
of times, and then applying $\log_q$ to the result.

First, let's make cons.

```
1   (define (cons-nnint a b)
2     (* (expt 2 a) (expt 3 b)))
3   (define (cons-nnint-debug a b)  ;; DEBUG
4     (let* ((aa (expt 2 a))
5           (bb (expt 3 b))
```

```
6          (ab (* aa bb)))
7      (display aa)
8      (newline)
9      (display bb)
10     (newline)
11     (display ab)
12     (newline)
13     ab))
```

Also, Guile doesn't have a function for custom logs so let's define that now.

```
1  (define (logn b p)
2    (/ (log p) (log b)))
```

Let's do some analysis to see how these numbers are related.

```
1  <<cons-nnint>>
2  (let*
3    ((tablesize 7)
4     (inputs (map (lambda(x)
5                     (map (lambda(y)
6                             (cons x y))
7                           (iota tablesize)))
8                   (iota tablesize)))
9     (outputs (map (lambda(row)
10                      (map (lambda(col)
11                              (cons-nnint (car col) (cdr col)))
12                            row))
13                   inputs)))
14   outputs)
```

1	3	9	27	81	243	729
2	6	18	54	162	486	1458
4	12	36	108	324	972	2916
8	24	72	216	648	1944	5832
16	48	144	432	1296	3888	11664
32	96	288	864	2592	7776	23328
64	192	576	1728	5184	15552	46656

Here are our scrambled numbers.

```
1  ;; To find a number of some base in some column,
2  ;; First divide by unwantedbase for targetcol number of times
3  <<repeated>>
4  (let ((targetcol 2)
5        (unwantedbase 3))
6    (map (lambda(row)
7           (map (lambda(item)
```

```
 8                ((repeated (lambda(x)
 9                            (/ x unwantedbase)) targetcol)
10              item))
11              row))
12        data))
```

$$
\begin{array}{ccccccc}
1/9 & 1/3 & 1 & 3 & 9 & 27 & 81 \\
2/9 & 2/3 & 2 & 6 & 18 & 54 & 162 \\
4/9 & 4/3 & 4 & 12 & 36 & 108 & 324 \\
8/9 & 8/3 & 8 & 24 & 72 & 216 & 648 \\
16/9 & 16/3 & 16 & 48 & 144 & 432 & 1296 \\
32/9 & 32/3 & 32 & 96 & 288 & 864 & 2592 \\
64/9 & 64/3 & 64 & 192 & 576 & 1728 & 5184 \\
\end{array}
$$

The numbers from our target column onwards are integers, with the target column being linearly exponentiated by 2 because the original numbers were linear.

```
1  <<logn>>
2  (let ((wantedbase 2))
3    (map (lambda(row)
4          (map (lambda(item)
5                (format #f "~6,3f" (logn 2 item)))
6                row))
7        data))
```

$$
\begin{array}{ccccccc}
-3.170 & -1.585 & 0.000 & 1.585 & 3.170 & 4.755 & 6.340 \\
-2.170 & -0.585 & 1.000 & 2.585 & 4.170 & 5.755 & 7.340 \\
-1.170 & 0.415 & 2.000 & 3.585 & 5.170 & 6.755 & 8.340 \\
-0.170 & 1.415 & 3.000 & 4.585 & 6.170 & 7.755 & 9.340 \\
0.830 & 2.415 & 4.000 & 5.585 & 7.170 & 8.755 & 10.340 \\
1.830 & 3.415 & 5.000 & 6.585 & 8.170 & 9.755 & 11.340 \\
2.830 & 4.415 & 6.000 & 7.585 & 9.170 & 10.755 & 12.340 \\
\end{array}
$$

Now the second column has recovered its original values. Although we didn't know what the original integer values were, we can now tell which column has the correct numbers by looking at which are integer values.

We can use this sign of a correct result in the proposed `car` and `cdr` procedures.

```
1  <<cons-nnint>>
2  <<logn>>
3  (use-srfis '(1))
4  (define (all-your-base ab unwanted wanted)
5    (if (equal? (modulo ab unwanted) 0)
6        (all-your-base (/ ab unwanted) unwanted wanted)
7        (if (equal? (modulo ab wanted) 0)
8            (round (logn wanted ab))
```

```
 9              "This number isn't a factor!")))
10    (define (car-nnint ab)
11      (all-your-base ab 3 2))
12    (define (cdr-nnint ab)
13      (all-your-base ab 2 3))
14
15    (let* ((initvalues '((2 3) (4 5) (7 2)))
16           (conslist (map (lambda(x)
17                            (apply cons-nnint x))
18                          initvalues))
19           (carlist (map (lambda(x)
20                           (car-nnint x))
21                         conslist))
22           (cdrlist (map (lambda(x)
23                           (cdr-nnint x))
24                         conslist)))
25      (map (lambda(x y) (cons x y))
26           (list "pairs" "cons'd" "car" "cdr")
27           (list initvalues conslist carlist cdrlist)))
```

pairs	(2 3)	(4 5)	(7 2)
cons'd	108	3888	1152
car	2.0	4.0	7.0
cdr	3.0	5.0	2.0

3.8   Exercise 2.6   optional

3.8.1   Question

In case representing pairs as procedures wasn't mind-boggling enough,
consider that, in a language that can manipulate procedures, we can
get by without numbers (at least insofar as nonnegative integers are
concerned) by implementing 0 and the operation of adding 1 as

```
1    (define zero (lambda (f) (lambda (x) x)))
2    (define (add-1 n)
3      (lambda (f) (lambda (x) (f ((n f) x)))))
```

This representation is known as *Church numerals*, after its inventor,
Alonzo Church, the logician who invented the λ-calculus.

Define one and two directly (not in terms of zero and add-1). (Hint:
Use substitution to evaluate (add-1 zero)). Give a direct definition
of the addition procedure + (not in terms of repeated application of
add-1).

3.8.2   Answer

First, let's check out (add-1 zero).

111

```
1   (define zero (lambda (f) (lambda (x) x)))
2   (define (add-1 n)
3     (lambda (f) (lambda (x)
4               (f ((n f) x)))))
5
6   (add-1 zero)
7   ((lambda (f) (lambda (x)
8               (f ((zero f) x)))))
9   ((lambda (f) (lambda (x)
10              (f ((lambda (x) x) x)))))
11  ((lambda (f) (lambda (x)
12              (f x))))
```

So from this I believe the correct definition of one and two are:

```
1   (load "mattcheck2.scm")
2   (define one
3     (lambda (f) (lambda (x)
4               (f x))))
5   (define two
6     (lambda (f) (lambda (x)
7                (f (f x)))))
8
9   (mattcheck "1 = 1+0"
10              1
11              ((one 1+) 0))
12  (mattcheck "2 = 1+1+0"
13              2
14              ((two 1+) 0))
15
16  (define (add a b)
17    (lambda (f) (lambda (x)
18              ((a f) ((b f) x)))))
19
20  (mattcheck "3 = 1+2 = (1+0) + (1+1+0)"
21              3
22              (((add one two) 1+) 0))
```

```
SUCCEED at 1 = 1+0
SUCCEED at 2 = 1+1+0
SUCCEED at 3 = 1+2 = (1+0) + (1+1+0)
```

3.9   Exercise 2.7

3.9.1   Text

```
1   (define (add-interval x y)
2     (make-interval (+ (lower-bound x) (lower-bound y))
3                    (+ (upper-bound x) (upper-bound y))))
```

```
4   (define (mul-interval x y)
5     (let ((p1 (* (lower-bound x) (lower-bound y)))
6           (p2 (* (lower-bound x) (upper-bound y)))
7           (p3 (* (upper-bound x) (lower-bound y)))
8           (p4 (* (upper-bound x) (upper-bound y))))
9       (make-interval (min p1 p2 p3 p4)
10                      (max p1 p2 p3 p4))))
11  (define (div-interval x y)
12    (mul-interval
13      x
14      (make-interval (/ 1.0 (upper-bound y))
15                     (/ 1.0 (lower-bound y)))))
```

3.9.2   Question

Alyssa's program is incomplete because she has not specified the implementation of the interval abstraction. Here is a definition of the interval constructor:

```
1   (define (make-interval a b) (cons a b))
```

Define selectors `upper-bound` and `lower-bound` to complete the implementation.

3.9.3   Answer

```
1   <<interval-txt>>
2
3   ;; Makes more sense to me to test
4   ;; order in the constructor than selector
5   (define (make-interval a b)
6     (if (> a b)
7         (cons a b)
8         (cons b a)))
9
10  (define (upper-bound i)
11    (car i))
12  (define (lower-bound i)
13    (cdr i))
```

3.10   Exercise 2.8

3.10.1   Question

Using reasoning analogous to Alyssa's, describe how the difference of two intervals may be computed. Define a corresponding subtraction procedure, called `sub-interval`.

3.10.2   Answer

I would argue that with one interval subtracted from the other, the lowest possible value is the lower of the first subtracted from the *upper* of the second, and the highest is the upper of the first subtracted from the lower of the second.

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

3.11   Exercise 2.9

3.11.1   Question

The *width* of an interval is half of the difference between its upper and lower bounds. The width is a measure of the uncertainty of the number specified by the interval. For some arithmetic operations the width of the result of combining two intervals is a function only of the widths of the argument intervals, whereas for others the width of the combination is not a function of the widths of the argument intervals. Show that the width of the sum (or difference) of two intervals is a function only of the widths of the intervals being added (or subtracted). Give examples to show that this is not true for multiplication or division.

3.11.2   Answer

My first interpretation of the question was that it asked whether width operations are *distributive*. For example, multiplication is distributive:

$$a(b + c) = (a \times b) + (a \times c)$$

For this I wrote the following tests:

```
(load "mattcheck2.scm")
<<make-interval>>
<<sub-interval>>

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

(define (width-interval I)
  (halve (- (upper-bound I)
            (lower-bound I))))

(let* ((ia (make-interval 10.1 9.9))
       (ib (make-interval 5.2 4.8))
       (Aab (add-interval ia ib))
```

```
15          (Sab (sub-interval ia ib))
16          (Mab (mul-interval ia ib))
17          (Dab (div-interval ia ib)))
18     (mattcheck-float "ia width = roughly .1"
19                       0.1
20                       (width-interval ia))
21     (mattcheck-float "ib width = roughly .2"
22                       0.2
23                       (width-interval ib))
24     (mattcheck-float "width addition is distributive"
25                       (width-interval Aab)
26                       (+ (width-interval ia)
27                          (width-interval ib)))
28     (mattcheck-float "width subtraction is distributive"
29                       (width-interval Sab)
30                       (- (width-interval ia)
31                          (width-interval ib)))
32     (mattcheck-float "width multiplication is distributive"
33                       (width-interval Mab)
34                       (* (width-interval ia)
35                          (width-interval ib)))
36     (mattcheck-float "width division is distributive"
37                       (width-interval Dab)
38                       (/ (width-interval ia)
39                          (width-interval ib))))
```

```
SUCCEED at ia width = roughly .1
SUCCEED at ib width = roughly .2
SUCCEED at width addition is distributive
FAIL at width subtraction is distributive
expected: -0.10000000000000053
returned: 0.2999999999999998
FAIL at width multiplication is distributive
expected: 0.01999999999999995
returned: 2.5
FAIL at width division is distributive
expected: 0.4999999999999978
returned: 0.10016025641025639
```

However upon rereading the question I see that it could be rephrased as "in what operations can you calculate the resulting interval's width with only the widths of the argument intervals?"

Basically, for argument interval x and y and result interval z:

```
IF z = x + y
THEN z_width = x_width + y_width
IF z = x - y
THEN z_width = x_width + y_width
```

$$\text{IF } z = x + y$$
$$\text{THEN } z_{width} = x_{width} + y_{width}$$
$$\text{IF } z = x - y$$
$$\text{THEN } z_{width} = x_{width} + y_{width}$$

Multiplied or divided widths cannot be determined from widths alone.

So, let's try that again.

```scheme
(load "mattcheck2.scm")
<<make-interval>>
<<sub-interval>>

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

(define (width-interval I)
  (halve (- (upper-bound I)
            (lower-bound I))))

(let* ((ia (make-interval 10.1 9.9))
       (ib (make-interval 5.2 4.8))
       (Aab (add-interval ia ib))
       (Sab (sub-interval ia ib)))
  (mattcheck-float "ia width = roughly .1"
                   0.1
                   (width-interval ia))
  (mattcheck-float "ib width = roughly .2"
                   0.2
                   (width-interval ib))
  (mattcheck-float "width(ia+ib) = width(ia) + width(ib)"
                   (width-interval Aab)
                   (+ (width-interval ia)
                      (width-interval ib)))
  (mattcheck-float "width(ia-ib) = width(ia) + width(ib)"
                   (width-interval Sab)
                   (+ (width-interval ia)
                      (width-interval ib))))
```

```
SUCCEED at ia width = roughly .1
SUCCEED at ib width = roughly .2
SUCCEED at width(ia+ib) = width(ia) + width(ib)
SUCCEED at width(ia-ib) = width(ia) + width(ib)
```

3.12   Exercise 2.10

3.12.1   Question

Ben Bitdiddle, an expert systems programmer, looks over Alyssa's shoulder and comments that it is not clear what it means to divide by an interval that spans zero. Modify Alyssa's code to check for this condition and to signal an error if it occurs.

116

3.12.2 Answer

```
1  (define (interval-spans-zero? I)
2    (and (> upper-bound 0)
3         (< lower-bound 0)))
4  (define (div-interval x y)
5    (if (interval-spans-zero? y)
6        "DIV-INTERVAL ERROR: denominator spans zero"
7        (mul-interval
8         x
9         (make-interval (/ 1.0 (upper-bound y))
10                        (/ 1.0 (lower-bound y))))))
```

3.13 Exercise 2.11

3.13.1 Question

In passing, Ben also cryptically comments: "By testing the signs
of the endpoints of the intervals, it is possible to break mul-int⌋
erval into nine cases, only one of which requires more than two
multiplications." Rewrite this procedure using Ben's suggestion.

3.13.2 Answer

This problem doesn't appear to have a beautiful, elegant answer.

Let's examine the nine cases.

```
1  (use-modules (ice-9 format))
2  (use-srfis '(1))
3  (load "mattcheck2.scm")
4  <<make-interval>>
5  <<sub-interval>>
6
7  (define (matt-examine-mult f)
8    (let* ((pp (make-interval 3 2))
9           (pn (make-interval 3 -5))
10          (pn2 (make-interval 1 -0.5))
11          (nn (make-interval -5 -7))
12          (listofpairs (list
13                        (list pp pp)
14                        (list pp pn)
15                        (list pp nn)
16                        (list pn pp)
17                        (list pn pn)
18                        (list pn pn2) ;; <- edge case to catch incomplete
19                        (list pn nn)  ;;    multiplication functions
20                        (list nn pp)
21                        (list nn pn)
22                        (list nn nn)))
```

```
23            (givesign (lambda(x)
24                         (if (negative? x)
25                             "-"
26                             "+")))
27            (print-sign (lambda(I)
28                          (format #f "~a ~a"
29                                  (givesign (upper-bound I))
30                                  (givesign (lower-bound I)))))
31            (print-int (lambda(I)
32                         (format #f "~a/~a"
33                                 (upper-bound I)
34                                 (lower-bound I))))
35            (print-ints (lambda(I J)
36                          (format #f "~a times ~a"
37                                  (print-int I)
38                                  (print-int J))))
39            (results (map (lambda(p)
40                            (apply f p))
41                          listofpairs)))
42       (list
43        (map (lambda(p)
44              (apply print-ints p))
45             listofpairs)
46        (map print-int results)
47        (map (lambda(I)
48              (print-sign I))
49             results)
50        (map (lambda(p)
51              (format #f "~a // ~a"
52                      (print-sign (car p))
53                      (print-sign (cadr p))))
54             listofpairs))))
55    (cons
56     (list "problem" "result" "signs" "problem signs")
57     (apply zip
58            (matt-examine-mult mul-interval)))
```

problem	result	signs	problem signs
3/2 times 3/2	9/4	+ +	+ + // + +
3/2 times 3/-5	9/-15	+ -	+ + // + -
3/2 times -5/-7	-10/-21	- -	+ + // - -
3/-5 times 3/2	9/-15	+ -	+ - // + +
3/-5 times 3/-5	25/-15	+ -	+ - // + -
3/-5 times 1/-0.5	3.0/-5.0	+ -	+ - // + -
3/-5 times -5/-7	35/-21	+ -	+ - // - -
-5/-7 times 3/2	-10/-21	- -	- - // + +
-5/-7 times 3/-5	35/-21	+ -	- - // + -
-5/-7 times -5/-7	49/25	+ +	- - // - -

```scheme
(define (mul-interval-opt x y)
  (let ((xu (upper-bound x))
        (xl (lower-bound x))
        (yu (upper-bound y))
        (yl (lower-bound y)))
    (define p? positive?)
    (define n? negative?)
    (define (check-signs? a b x y) ;; pass functions
      (and (a xu)
           (b xl)
           (x yu)
           (y yl)))
    (define (same-signs?)
      (or (check-signs? p? p? p? p?)
          (check-signs? n? n? n? n?)))
    (define (alt-signs?)
      (or (check-signs? p? p? n? n?)
          (check-signs? n? n? p? p?)))
    (cond ((same-signs?)
           (make-interval (* xu yu)
                          (* xl yl)))
          ((alt-signs?)
           (make-interval (* xl yu)
                          (* xu yl)))
          ((check-signs? p? p? p? n?)
           (make-interval (* xu yu)
                          (* xu yl)))
          ((check-signs? p? n? p? p?)
           (make-interval (* xu yu)
                          (* xl yu)))
          ((check-signs? p? n? p? n?)
           (let ((p1 (* xu yu))
                 (p2 (* xu yl))
                 (p3 (* xl yu))
                 (p4 (* xl yl)))
             (make-interval (max p1 p2 p3 p4)
                            (min p1 p2 p3 p4))))
          ((check-signs? p? n? n? n?)
           (make-interval (* xl yl)
                          (* xu yl)))
          ((check-signs? n? n? p? n?)
           (make-interval (* xl yl)
                          (* xl yu))))))
```

```scheme
(use-modules (ice-9 format))
(use-srfis '(1))
(load "mattcheck2.scm")
```

```
 4   <<make-interval>>
 5   <<sub-interval>>
 6   <<mul-interval-opt>>
 7
 8   (define (matt-mult-consistency f1 f2)
 9     (let* ((pp (make-interval 3 2))
10           (pn (make-interval 3 -5))
11           (pn2 (make-interval 1 -0.5))
12           (nn (make-interval -5 -7))
13           (listofpairs (list
14                         (list "pp*nn" pp pp)
15                         (list "pp*pn" pp pn)
16                         (list "pp*nn" pp nn)
17                         (list "pn*pp" pn pp)
18                         (list "pn*pn" pn pn)
19                         (list "pn*pn2" pn pn2) ;; <- edge case to catch
                          ↪  incomplete
20                         (list "pn*nn" pn nn)  ;;    multiplication
                          ↪  functions
21                         (list "nn*pp" nn pp)
22                         (list "nn*pn" nn pn)
23                         (list "nn*nn" nn nn))))
24       (map (lambda(l)
25             (mattcheck (car l)
26                        (apply f1 (cdr l))
27                        (apply f2 (cdr l))))
28           listofpairs)))
29
30   (matt-mult-consistency mul-interval mul-interval-opt)
```

```
SUCCEED at pp*nn
SUCCEED at pp*pn
SUCCEED at pp*nn
SUCCEED at pn*pp
SUCCEED at pn*pn
SUCCEED at pn*pn2
SUCCEED at pn*nn
SUCCEED at nn*pp
SUCCEED at nn*pn
SUCCEED at nn*nn


Unoptimized mul-interval: (5231.8421225)
Optimized mul-interval: (2526.5896437)
```

So as expected, about twice as fast!

3.14 Exercise 2.12

3.14.1 Question

After debugging her program, Alyssa shows it to a potential user, who complains that her program solves the wrong problem. He wants a program that can deal with numbers represented as a center value and an additive tolerance; for example, he wants to work with intervals such as 3.5 ± 0.15 rather than [3.35, 3.65]. Alyssa returns to her desk and fixes this problem by supplying an alternate constructor and alternate selectors:

```
(define (make-center-width c w)
  (make-interval (- c w) (+ c w)))
(define (center i)
  (/ (+ (lower-bound i) (upper-bound i)) 2))
(define (width i)
  (/ (- (upper-bound i) (lower-bound i)) 2))
```

Unfortunately, most of Alyssa's users are engineers. Real engineering situations usually involve measurements with only a small uncertainty, measured as the ratio of the width of the interval to the midpoint of the interval. Engineers usually specify percentage tolerances on the parameters of devices, as in the resistor specifications given earlier.

Define a constructor `make-center-percent` that takes a center and a percentage tolerance and produces the desired interval. You must also define a selector `percent` that produces the percentage tolerance for a given interval. The `center` selector is the same as the one shown above.

3.14.2 Answer

```
(define (reciprocal x)
  (/ 1 x))
```

```
<<reciprocal>>
<<make-interval>>
<<sub-interval>>
<<interval-center-width>>

(define (make-center-percent c pt)
  (let ((pp (* c
              (* pt 0.01))))
    (make-interval (- c pp) (+ c pp))))
(define (percent I)
  (* 100.0
     (/ (width I)
        (center I))))
```

```
1   <<interval-percent>>
2   (load "mattcheck2.scm")
3     (define (roughly-eq? a b)
4       ;; error size varies with magnitude of fp
5       ;; so dx must vary too.
6       (define dx (* a 0.000001))
7       (and (> a (- b dx))
8            (< a (+ b dx))))
9   (define (interval-roughly-eq? I J)
10    (and (roughly-eq? (upper-bound I) (upper-bound J))
11         (roughly-eq? (lower-bound I) (lower-bound J))))
12
13  (let* ((i1 (make-interval 105.0 95.0))
14         (i2 (make-center-width 100.0 5))
15         (i3 (make-center-percent 100.0 5))
16         (i1a (upper-bound i1)))
17    (mattcheck "make-center-width"
18               i1
19               i2)
20    (mattcheck "make-center-percent"
21               i1
22               i3)
23    (mattcheck "percent"
24               (percent i1)
25               (percent i3)
26               5.0)
27    (mattcheck+ "make-center-percent is consistent"
28                (list i1 i3)
29                #:eql? interval-roughly-eq?))
```

```
SUCCEED at make-center-width
SUCCEED at make-center-percent
SUCCEED at percent
SUCCEED at make-center-percent is consistent
```

3.15   Exercise 2.13   optional

3.15.1   Question

> Show that under the assumption of small percentage tolerances there
> is a simple formula for the approximate percentage tolerance of the
> product of two intervals in terms of the tolerances of the factors.
> You may simplify the problem by assuming that all numbers are
> positive.

3.15.2   Answer

I should've written this function a while ago.

```scheme
(use-modules (ice-9 format))
(define (stringit . args)
  (string-append
    (apply string-append
      (map (lambda(x)
             (format #f "~:a " x))
           args))))
(define (echo . args)
  (format #t "~&~:a~%" (apply stringit args)))
```

Now, let's examine how interval percents relate to each other.

```scheme
<<echo>>
<<interval-percent>>
<<mul-interval-opt>>

(let* ((i1 (make-center-width 100 5))
       (i2 (make-center-width 200 5))
       (M12 (mul-interval-opt i1 i2)))
  (echo "intervals 1 and 2:" i1 i2)
  (echo "width of 1 and 2:" (width i1) (width i2))
  (echo "percent of 1 and 2:" (percent i1) (percent i2))
  (echo "i1*i2 = " M12)
  (echo "width M12:" (width M12))
  (echo "percent M12:" (percent M12)))
```

```
intervals 1 and 2: (105 . 95) (205 . 195)
width of 1 and 2: 5 5
percent of 1 and 2: 5.0 2.5
i1*i2 =  (21525 . 18525)
width M12 1500
percent M12 7.490636704119851
```

Perhaps percent$(A \times B) =$percent$(A)+$percent(B)?

```scheme
<<echo>>
<<interval-percent>>
<<mul-interval-opt>>

(let* ((i1 (make-center-percent 40 0.1))
       (i2 (make-center-percent 200 0.4))
       (M12 (mul-interval-opt i1 i2)))
  (echo "percent of 1 and 2:" (percent i1) (percent i2))
  (echo "percent M12:" (percent M12)))
```

```
percent of 1 and 2: 0.09999999999999788 0.40000000000000563
percent M12: 0.499998000008
```

3.16 Exercise 2.14

3.16.1 Question

After considerable work, Alyssa P. Hacker delivers her finished system. Several years later, after she has forgotten all about it, she gets a frenzied call from an irate user, Lem E. Tweakit. It seems that Lem has noticed that the formula for parallel resistors can be written in two algebraically equivalent ways:

$$\frac{R_1 R_2}{R_1 + R_2}$$

and

$$\frac{1}{\frac{1}{R_1} + \frac{1}{R_2}}$$

He has written the following two programs, each of which computes the parallel-resistors formula differently:

```
1  (define (par1 r1 r2)
2    (div-interval (mul-interval r1 r2)
3                  (add-interval r1 r2)))
4
5  (define (par2 r1 r2)
6    (let ((one (make-interval 1 1)))
7      (div-interval
8        one (add-interval (div-interval one r1)
9                          (div-interval one r2)))))
```

Lem complains that Alyssa's program gives different answers for the two ways of computing. This is a serious complaint.

Demonstrate that Lem is right. Investigate the behavior of the system on a variety of arithmetic expressions. Make some intervals A and B, and use them in computing the expressions A/A and A/B. You will get the most insight by using intervals whose width is a small percentage of the center value. Examine the results of the computation in center-percent form (see Exercise 2.12).

3.16.2 Answer

```
1  <<echo>>
2  <<interval-percent>>
3  <<mul-interval-opt>>
4  <<par-resistors>>
```

```scheme
   (let* ((A (make-center-percent 10 1))
          (B (make-center-percent 10 0.01))
          (p1 (par1 A B))
          (p2 (par2 A B)))
     (echo "A,B:" A B)
     (echo "par1(A,B):" p1)
     (echo "par2(A,B):" p2)
     (echo "percent(par1):" (percent p1))
     (echo "percent(par2):" (percent p2))
     (echo "center(par1):" (center p1))
     (echo "center(par2):" (center p2)))

   (echo "So these two have inconsistent effects on the width.")
   (newline)
   (echo "It should also be noted that floating-point errors accumulate.")
   (echo "Take a look at the error on these (correct answer is 1)")

   (let* ((A (make-center-percent 10 1))
          (p1 (div-interval
               (div-interval
                (mul-interval A A)
                A)
               A))
          (p2 (div-interval
               (div-interval
                (div-interval
                 (mul-interval
                  (mul-interval A A)
                  A)
                 A)
                A)
               A)))
     (echo "p1:" (center p1))
     (echo "p2:" (center p2)))
```

```
A,B: (10.1 . 9.9) (10.001 . 9.999)
par1(A,B): (5.076139504497713 . 4.924635590269141)
par2(A,B): (5.025128103079449 . 4.974626865671642)
percent(par1): 1.5149217214958663
percent(par2): 0.5050247487625606
center(par1): 5.000387547383427
center(par2): 4.999877484375546
So these two have inconsistent effects on the width.

It should also be noted that floating-point errors accumulate.
Take a look at the error on these (correct answer is 1)
p1: 1.0008001600240033
p2: 1.0018006601460259
```

3.17   Exercise 2.15

3.17.1   Question

Eva Lu Ator, another user, has also noticed the different intervals computed by different but algebraically equivalent expressions. She says that a formula to compute with intervals using Alyssa's system will produce tighter error bounds if it can be written in such a form that no variable that represents an uncertain number is repeated. Thus, she says, `par2` is a "better" program for parallel resistances than `par1`. Is she right? Why?

3.17.2   Answer

If I am correct in understanding that "uncertain number" means "a number with an error tolerance", than `par2` *is* better – it only uses two instances of variables with error tolerance, while `par1` uses four.

It should be noted that this system does not directly translate to algebraic expressions. For example, take these expressions:

$$A + A = 2A$$

$$A - A = 0$$

$$A/A = 1$$

Note that these do not hold up in practice with uncertain numbers:

```
1    <<echo>>
2    <<interval-percent>>
3    <<mul-interval-opt>>
4
5    (define A (make-center-percent 10 1))
6    (echo "A+A = 2A ≠" (add-interval A A))
7    (echo "A-A = 0 ≠" (sub-interval A A))
8    (echo "A/A = 1 ≠" (div-interval A A))
```

```
A+A = 2A ≠ (20.2 . 19.8)
A-A = 0 ≠ (0.1999999999999993 . -0.1999999999999993)
A/A = 1 ≠ (1.02020202020202 . 0.9801980198019803)
```

3.18   Exercise 2.16   optional

3.18.1   Question

Explain, in general, why equivalent algebraic expressions may lead to different answers. Can you devise an interval-arithmetic package that does not have this shortcoming, or is this task impossible? (Warning: This problem is very difficult.)

3.18.2 Answer

It is *indeed* very difficult, because from what I'm seeing online, no interval system without these issues exists. To avoid these issues, interval mathematics would need to satisfy the conditions for a **field** – and failing that, needs to only use each variable once, which becomes impossible as soon as you encounter an expression as simple as x^2.

GitHub user "diiq" has an incredible analysis of this, which can be found here: `https://gist.github.com/diiq/1f39df0e54b2137bb07e7e04b11cb075`

3.19   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)
```
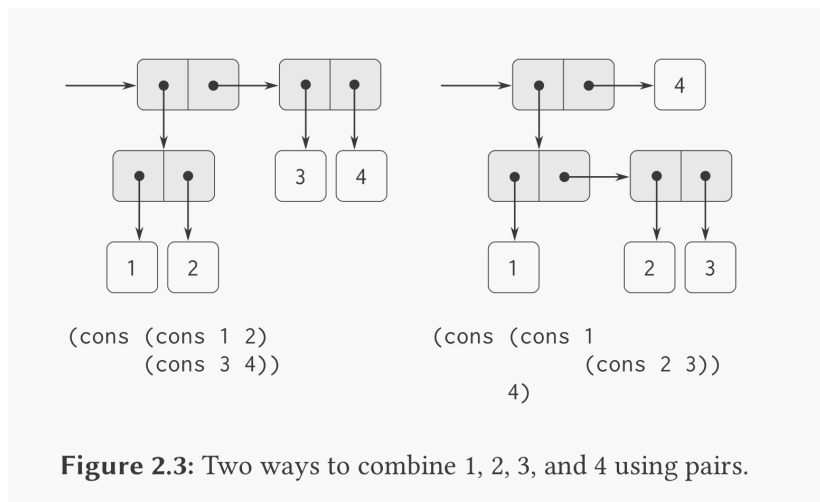
Figure 2.3: Two ways to combine 1, 2, 3, and 4 using pairs.

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.

3.20   2.2.1: Representing Sequences

sequence An ordered collection of data objects.

list A sequence of `cons` pairs.

```
(cons 1
      (cons 2
            (cons 3
                  (cons 4 nil))))
(list 1 2 3 4)
;; both evaluate to '(1 2 3 4)
```

An aside: many parts of this book have covered ways to solve problems by splitting problems into simple recursive solutions. I may be getting ahead of myself, but I wanted to note how the `cons` pair system goes hand-in-hand with this. For example, when going over a list `l` with function `f`:

```
(define (map f l)
  (if (null? l)
      #nil
      (cons (f (car l))
            (map f (cdr l)))))

(map (lambda(x)(* x 2))
     (list 1 2 3 4))
```

| 2 | 4 | 6 | 8 |

3.21   Exercise 2.17

3.21.1   Question

Define a procedure `last-pair` that returns the list that contains only the last element of a given (nonempty) list:

```
(last-pair (list 23 72 149 34))
;; (34)
```

3.21.2   Answer

```
(define (last-pair l)
  (let ((a (car l))
        (d (cdr l)))
    (if (= 1 (length d))
        d
```

```
6            (last-pair d))))
```

```
1  <<last-pair>>
2  (last-pair (list 23 72 149 34))
```

| 34 |

3.22 Exercise 2.18

3.22.1 Question

Define a procedure `reverse` that takes a list as argument and returns
a list of the same elements in reverse order:

```
1  (reverse (list 1 4 9 16 25))
2  ;; (25 16 9 4 1)
```

3.22.2 Answer

```
1  (define (reverse l)
2    (define len (length l))
3    (define (iter i result)
4      (if (< (1- len) i)
5          result
6          (iter (+ i 1)
7                (cons (list-ref l i)
8                      result))))
9    (iter 0 '()))
```

```
1  <<reverse>>
2  (reverse (list 23 72 149 34))
```

34   149   72   23

3.23 Exercise 2.19

3.23.1 Question

Consider the change-counting program of 1.2.2. It would be nice to
be able to easily change the currency used by the program, so that
we could compute the number of ways to change a British pound, for
example. As the program is written, the knowledge of the currency is
distributed partly into the procedure `first-denomination` and partly
into the procedure `count-change` (which knows that there are five

kinds of U.S. coins). It would be nicer to be able to supply a list of coins to be used for making change.

We want to rewrite the procedure `cc` so that its second argument is a list of the values of the coins to use rather than an integer specifying which coins to use. We could then have lists that defined each kind of currency:

```
(define us-coins (list 50 25 10 5 1))
(define uk-coins (list 100 50 20 10 5 2 1 0.5))
```

We could then call `cc` as follows:

```
(cc 100 us-coins)
; 292
```

To do this will require changing the program `cc` somewhat. It will still have the same form, but it will access its second argument differently, as follows:

```
(define (cc amount coin-values)
  (cond ((= amount 0) 1)
        ((or (< amount 0) (no-more? coin-values)) 0)
        (else
         (+ (cc amount
                (except-first-denomination
                 coin-values))
            (cc (- amount
                   (first-denomination
                    coin-values))
                coin-values)))))
```

Define the procedures `first-denomination`, `except-first-denomination`, and `no-more?` in terms of primitive operations on list structures. Does the order of the list `coin-values` affect the answer produced by `cc`? Why or why not?

3.23.2 Answer

```
<<cc-lists>>

(define us-coins
  (list 50 25 10 5 1))
(define uk-coins
  (list 100 50 20 10 5 2 1 0.5))

(define first-denomination car)
```

130

```
 9  (define except-first-denomination cdr)
10  (define no-more? null?)
```

```
1  <<Ex-2-19>>
2  (list
3   (cc 100 us-coins)
4   (cc 100 (reverse us-coins))
5   (cc 100 (list 50 10 25 5 1)))
```

| 292 | 292 | 292 |

Apparently, the order of the list does *not* affect the value. However, it does effect the execution time, with small-to-large coin lists taking more time than large-to-small.

```
decreasing values: (357503.80704)
increasing values: (823460.64376)
```

3.24   Exercise 2.20

3.24.1   Question

Use [dotted-pair] notation to write a procedure same-parity that takes one or more integers and returns a list of all the arguments that have the same even-odd parity as the first argument. For example,

```
1  (same-parity 1 2 3 4 5 6 7)
2  ; (1 3 5 7)
3  (same-parity 2 3 4 5 6 7)
4  ; (2 4 6)
```

3.24.2   Answer

```
 1  (define (same-parity . rest)
 2    (define same?
 3      (if (even? (car rest))
 4          even?
 5          odd?))
 6    (define (iter l results)
 7      (if (null? l)
 8          results
 9          (let ((a (car l)))
10            (iter (cdr l)
11                  (if (same? a)
12                      (cons a results)
13                      results)))))
14    (iter (reverse rest) '()))
```

131

```
15
16   ;; Attempting to remove the reversing
17   (define (same-parity2 . args)
18     (define first (car args))
19     (define same?
20       (if (even? first)
21            even?
22            odd?))
23     (define (iter l results)
24       (if (null? l)
25            results
26            (let ((a (car l))
27                  (d (cdr l)))
28              (if (same? a)
29                  (iter d (append results
30                                   (cons a #nil)))
31                  (iter d results)))))
32     (iter (cdr args) (cons first #nil)))
```

```
1   <<same-parity>>
2   (list
3    (same-parity 1 2 3 4 5 6 7)
4    (same-parity2 2 3 4 5 6 7))
```

```
| 1 | 3 | 5 | 7 |
| 2 | 4 | 6 |   |

same-parity: (10003.483436)
same-parity2: (56007.042334)
```

Once again, my attempts to optimize are a complete failure. I'm guessing that the act of traversing the whole list in the call to append is the problem.

3.25   Exercise 2.21

3.25.1   Question

The procedure square-list takes a list of numbers as argument and returns a list of the squares of those numbers.

```
1   (square-list (list 1 2 3 4))
2   ;; (1 4 9 16)
```

Here are two different definitions of square-list. Complete both of them by filling in the missing expressions:

132

```
1  (define (square-list items)
2    (if (null? items)
3        nil
4        (cons <??> <??>)))
5  (define (square-list items)
6    (map <??> <??>))
```

3.25.2 Answer

```
1  <<square>>
2  (define (square-list-manual items)
3    (if (null? items)
4        #nil
5        (cons (square (car items))
6              (square-list-manual (cdr items)))))
7  (define (square-list items)
8    (map square items))
```

```
1  <<square-list>>
2  (let ((l (list 2 3 4 5 6)))
3    (list l
4          (square-list-manual l)
5          (square-list l)))
```

```
| 2 | 3 |  4 |  5 |  6 |
| 4 | 9 | 16 | 25 | 36 |
| 4 | 9 | 16 | 25 | 36 |
```

3.26   Exercise 2.22

3.26.1   Questions

Louis Reasoner tries to rewrite the first `square-list` procedure of
Exercise 2.21 so that it evolves an iterative process:

```
1  (define (square-list items)
2    (define (iter things answer)
3      (if (null? things)
4          answer
5          (iter (cdr things)
6                (cons (square (car things))
7                      answer))))
8    (iter items nil))
```

Unfortunately, defining `square-list` this way produces the answer
list in the reverse order of the one desired. Why?

133

Louis then tries to fix his bug by interchanging the arguments to `cons`:

```
1  (define (square-list items)
2    (define (iter things answer)
3      (if (null? things)
4          answer
5          (iter (cdr things)
6                (cons answer
7                      (square (car things))))))
8    (iter items nil))
```

This doesn't work either. Explain.

3.26.2   Answer

I'm positive I've made this exact mistake before, though this is likely not recorded.

The first form of `square-list` produces a correct list in reverse order:

```
1  (square-list (iota 6))
2  (25 16 9 4 1 0)
```

This is because he is prepending to the list every iteration.

While the second produces a broken list, which is literally backwards:

```
1  (square-list (iota 6))
2  ((((((#nil . 0) . 1) . 4) . 9) . 16) . 25)
3  ;; Equivalent to:
4  (cons (cons (cons (cons (cons (cons #nil
5                                      0)
6                                1)
7                          4)
8                    9)
9              16)
10        25)
```

Since Lisp was designed with the `cons pair` structure of list-building, it needed to define a "correct" direction for the pairs to go. Since the Western world thinks left-to-right, they made it so that the left (first) cell is for content, and the right is for the pointer to the next pair. However, this means that you can't append to a list without first traveling its length and changing the `nil` marking the end to a pointer to your new pair. Since that is a lot of list traveling, it makes more sense to `cons` your list together in reverse and then calling `reverse` only once at the end of the procedure.

134

3.27 Exercise 2.23

3.27.1 Question

The procedure `for-each` is similar to `map`. It takes as arguments a procedure and a list of elements. However, rather than forming a list of the results, `for-each` just applies the procedure to each of the elements in turn, from left to right. The values returned by applying the procedure to the elements are not used at all—`for-ea`⌋ `ch` is used with procedures that perform an action, such as printing. For example,

```
1  (for-each (lambda (x)
2             (newline)
3             (display x))
4           (list 57 321 88))
5  ;; 57
6  ;; 321
7  ;; 88
```

The value returned by the call to `for-each` (not illustrated above) can be something arbitrary, such as true. Give an implementation of `for-each`.

3.27.2 Answer

```
1  (define (for-each-mine proc items)
2    (define (iter l)
3      (if (null? l)
4          #t
5          (begin (proc (car l))
6                 (iter (cdr l)))))
7    (iter items))
```

```
1  <<for-each-mine>>
2  (for-each-mine (lambda(x)(display x)(display " ")) (list "all" "your"
   ↪  "base"))
3  (for-each (lambda(x)(display x)(display " ")) (list "are" "belong" "to"
   ↪  "us"))
```

```
all your base are belong to us
```

3.28 Exercise 2.24

3.28.1 Text Definitions

```
1  (define (count-leaves x)
2    (cond ((null? x) 0)
3          ((not (pair? x)) 1)
4          (else (+ (count-leaves (car x))
5                   (count-leaves (cdr x))))))
```

3.28.2 Question

Suppose we evaluate the expression (list 1 (list 2 (list 3 4))).
Give the result printed by the interpreter, the corresponding box-
and-pointer structure, and the interpretation of this as a tree (as in
Figure 2.6).

3.28.3 Answer

This is sort of a trick question – on first reading, I read it like a series of cons
statements. Looking again, though, I can see that the correct formulation is as
follows:

```
1   <<echo>>
2   (let ((l1 (list 1 (list 2 (list 3 4))))
3         (l2 (cons 1
4                   (cons
5                    (cons 2
6                          (cons
7                           (cons 3
8                                 (cons 4
9                                       #nil))
10                          #nil))
11                   #nil))))
12     (echo "textbook version:" l1)
13     (echo "cons'd version:" l2))
```
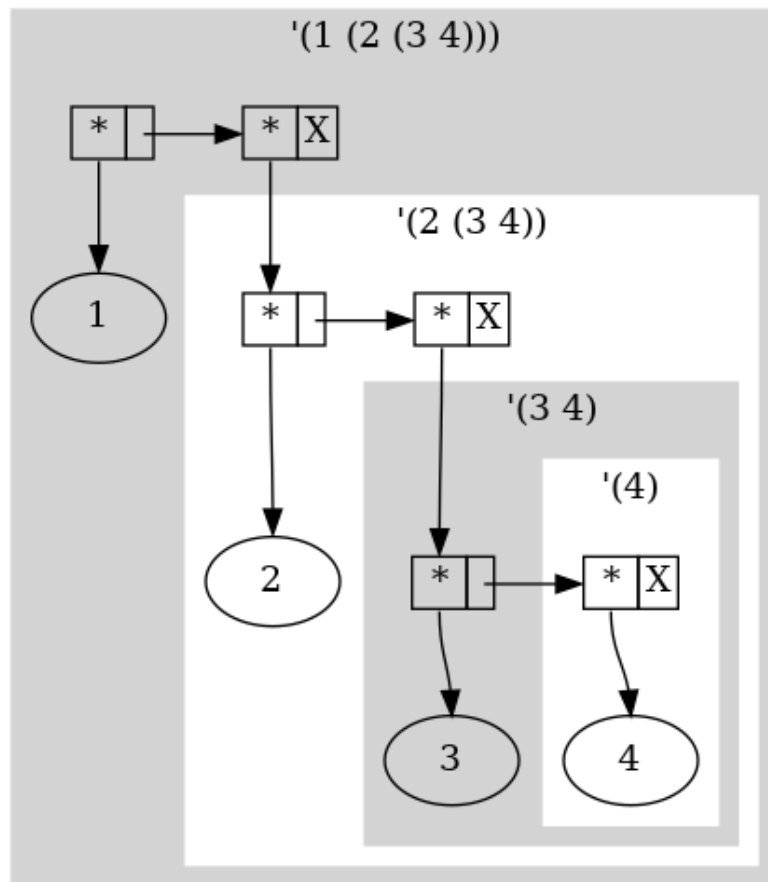
```
textbook version: (1 (2 (3 4)))
cons'd version: (1 (2 (3 4)))
```
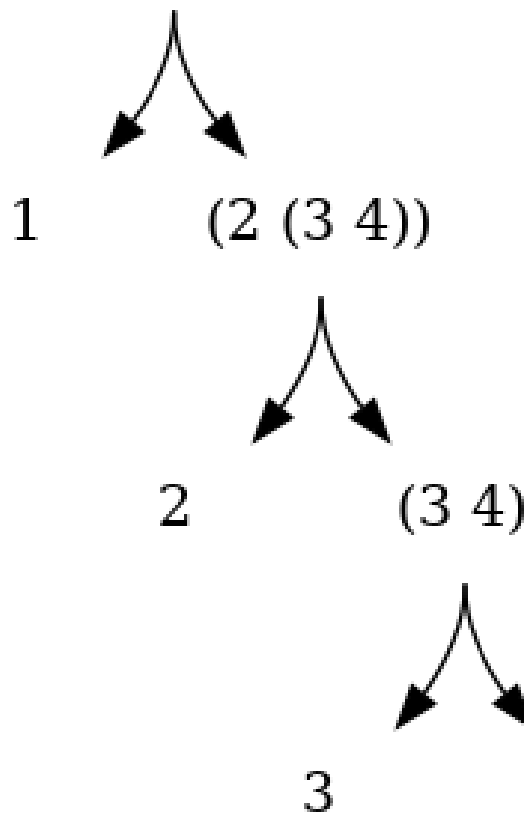
Dot and box version:

136

'(1 (2 (3 4)))

'(2 (3 4))

'(3 4)

'(4)

Tree version:

$$(1\ (2\ (3\ 4)))$$

$$1 \qquad (2\ (3\ 4))$$

$$2 \qquad (3\ 4)$$

$$3 \qquad 4$$

3.29   Exercise 2.25

3.29.1   Question

Give combinations of `car`s and `cdr`s that will pick 7 from each of the following lists:

```
(1 3 (5 7) 9)
((7))
(1 (2 (3 (4 (5 (6 7))))))
```

```
1  <<echo>>
2  (let ((l1 (list 1 3 (list 5 7) 9))
3        (l2 (list (list 7)))
4        (l3 (list 1 (list 2 (list 3 (list 4 (list 5 (list 6 7))))))))
5    (echo (car (cdaddr l1))
6          (caar l2)
7          (cadadr (cadadr (cadadr l3)))))
```

```
7 7 7
```

3.30 Exercise 2.26

3.30.1 Question

Suppose we define x and y to be two lists:

```
1  (define x (list 1 2 3))
2  (define y (list 4 5 6))
```

What result is printed by the interpreter in response to evaluating each of the following expressions:

```
1  (append x y)
2  (cons x y)
3  (list x y)
```

3.30.2 Answer

```
1  <<echo>>
2  (let* ((x (list 1 2 3))
3         (y (list 4 5 6))
4         (e1 (append x y))
5         (e2 (cons x y))
6         (e3 (list x y)))
7    (echo "(append x y):" e1)
8    (echo "(cons x y):" e2)
9    (echo "(list x y):" e3))
```

```
(append x y): (1 2 3 4 5 6)
(cons x y): ((1 2 3) 4 5 6)
(list x y): ((1 2 3) (4 5 6))
```

139

3.31 Exercise 2.27

3.31.1 Question

Modify your `reverse` procedure of Exercise 2.18 to produce a `deep-reverse` procedure that takes a list as argument and returns as its value the list with its elements reversed and with all sublists deep-reversed as well. For example,

```
(define x (list (list 1 2) (list 3 4)))
x
;; ((1 2) (3 4))
(reverse x)
;; ((3 4) (1 2))
(deep-reverse x)
;; ((4 3) (2 1))
```

3.31.2 Answer

```
(define (deep-reverse l)
  (define len (length l))
  (define (iter i result)
    (if (< (1- len) i)
        result
        (iter (+ i 1)
              (cons (let ((here (list-ref l i)))
                      (if (pair? here)
                          (deep-reverse here)
                          here))
                    result))))
  (iter 0 '()))
```

```
<<deep-reverse>>
(deep-reverse (list (list 1 2) (list 3 4)))
```

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

3.32 Exercise 2.28

3.32.1 Question

Write a procedure `fringe` that takes as argument a tree (represented as a list) and returns a list whose elements are all the leaves of the tree arranged in left-to-right order. For example,

```
(define x (list (list 1 2) (list 3 4)))
(fringe x)
;; (1 2 3 4)
```

```
4   (fringe (list x x))
5   ;; (1 2 3 4 1 2 3 4)
```

3.32.2   Answer

```
1   (define (fringe l)
2     (if (null? l)
3         #nil
4         (let ((a (car l))
5               (d (cdr l)))
6           (append (if (pair? a)
7                       (fringe a)
8                       (list a))
9                   (fringe d)))))
```

```
1   <<fringe>>
2   (fringe (list (list 1 (list 2 3)) (list 4 5)))
```

(1 2 3 4 5)

3.33   Exercise 2.29: Binary Mobiles

3.33.1   Text Definitions

```
1   (define (make-mobile left right)
2     (list left right))
3
4   (define (make-branch length structure)
5     (list length structure))
```

3.33.2   Question A: Selectors

Write the corresponding selectors `left-branch` and `right-branch`, which return the branches of a mobile, and `branch-length` and `br⌋ anch-structure`, which return the components of a branch.

3.33.3   Answer A

```
1   <<mobile-constructors-list>>
2   (define (left-branch mobile)
3     (car mobile))
4   (define (right-branch mobile)
5     (cadr mobile))
6   (define (branch-length branch)
7     (car branch))
```

```
8    (define (branch-structure branch)
9      (cadr branch))
```

3.33.4   Question B: total-weight

Using your selectors, define a procedure `total-weight` that returns
the total weight of a mobile.

3.33.5   Answer B

```
1    (define (total-weight mobile)
2      (let ((leftS (branch-structure (left-branch mobile)))
3            (rightS (branch-structure (right-branch mobile))))
4        (+ (if (number? leftS)
5               leftS
6               (total-weight leftS))
7           (if (number? rightS)
8               rightS
9               (total-weight rightS)))))
```

```
1    <<mobile-selectors-list>>
2    <<mobile-total-weight>>
3
4    (let ((M1 (make-mobile
5               (make-branch 5 5)
6               (make-branch 1
7                            (make-mobile (make-branch 2 2)
8                                         (make-branch 2 3)))))
9          (M2 (make-mobile
10              (make-branch 2
11                           (make-mobile
12                            (make-branch 2 2)
13                            (make-branch 2 2)))
14              (make-branch 2
15                           (make-mobile
16                            (make-branch 2 2)
17                            (make-branch 2 2))))))
18     (list (total-weight M1)
19           (total-weight M2)))
```

| 10 | 8 |

3.33.6   Question C: Balancing

A mobile is said to be **balanced** if the torque applied by its top-left branch is
equal to that applied by its top-right branch (that is, if the length of the left rod
multiplied by the weight hanging from that rod is equal to the corresponding

product for the right side) and if each of the submobiles hanging off its branches is balanced. Design a predicate that tests whether a binary mobile is balanced.

3.33.7   Answer C

I can imagine a ton of ways I could shoot myself in the foot by starting with optimization, so let's just try to nail down exactly what needs to happen.

```
1   (define (total-torque branch)
2     (let ((len (branch-length branch))
3           (struct (branch-structure branch)))
4       (* len
5          (if (number? struct)
6              struct
7              (let ((lbs (branch-structure (left-branch struct)))
8                    (rbs (branch-structure (right-branch struct))))
9                (+ (total-weight lbs)
10                  (total-weight rbs)))))))
11  (define (balanced? mobile)
12    (let* ((l (left-branch mobile))
13           (ls (branch-structure l))
14           (r (right-branch mobile))
15           (rs (branch-structure r))
16           (l-balanced (if (number? ls)
17                           #t
18                           (balanced? ls)))
19           (r-balanced (if (number? rs)
20                           #t ;; Fixed: accidentally used ls again.
21                           (balanced? rs))))
22      (if (and l-balanced r-balanced)
23          (= (total-torque l)
24             (total-torque r))
25          #f)))
```

I'll also need a modified `total-weight` that can notice when its argument is a non-mobile and just return the value.

```
1   (define (total-weight mobile)
2     (if (number? mobile)
3         mobile ;; this is a weight, just return it
4         (let ((leftS (branch-structure (left-branch mobile)))
5               (rightS (branch-structure (right-branch mobile))))
6           (+ (if (number? leftS)
7                  leftS
8                  (total-weight leftS))
9              (if (number? rightS)
10                 rightS
11                 (total-weight rightS))))))
```

143

```scheme
<<echo>>
<<mobile-selectors-list>>
<<mobile-total-weight-2>>
<<mobile-balanced-dumb>>
(define M1 ;; all segments unbalanced
  (make-mobile
   (make-branch 5 5) ;; torque 25 = 5*5
   (make-branch 1 ;; torque 5 = (2 + 3) * 1
                (make-mobile (make-branch 2 2)
                             (make-branch 2 3)))))
(define M2 ;; all segments balanced as they are duplicates
  (make-mobile
   (make-branch 2 ;; torque 8 = 2*(2+2)
                (make-mobile
                 (make-branch 2 2)
                 (make-branch 2 2)))
   (make-branch 2
                (make-mobile
                 (make-branch 2 2)
                 (make-branch 2 2)))))
(define M3 ;; equal torque, but one segment is unbalanced.
  (make-mobile
   (make-branch 2 ;; torque 12 = 2*(4+2)
                (make-mobile
                 (make-branch 2 4) ;; torque 8 = 2*4
                 (make-branch 4 2))) ;torque 8 = 4*2
   (make-branch 3 ;; torque 12 = 3*(1+3)
                (make-mobile
                 (make-branch 1 1) ;; torque 1
                 (make-branch 1 3))))) ;; torque 3
(define M4 ;; equal torque
  (make-mobile
   (make-branch 2 ;; torque 12 = 2*(4+2)
                (make-mobile
                 (make-branch 2 4) ;; torque 8 = 2*4
                 (make-branch 4 2))) ;torque 8 = 4*2
   (make-branch 3 ;; torque 12 = 3*(1+3)
                (make-mobile
                 (make-branch 3 1) ;; torque 3
                 (make-branch 1 3))))) ;; torque 3
(define (isbalanced? Name Status)
  (define Success "is balanced!")
  (define Failure "is not balanced!")
  (if (eq? Status #t)
      (echo Name Success)
      (echo Name Failure)))
(isbalanced? "M1" (balanced? M1))
(isbalanced? "M2" (balanced? M2))
```

```
49    (isbalanced? "M3" (balanced? M3))
50    (isbalanced? "M4" (balanced? M4))
```

```
M1 is not balanced!
M2 is balanced!
M3 is not balanced!
M4 is balanced!
```

This one took quite some fiddling. First I struggled to figure out exactly how I should juggle of torque, weight, and balance. For example, a mobile is balanced if the torques of its branches are equal, and if every submobile is also balanced, with torque being defined as length×weight. Note that it's the *weight*, not its submobile's *torque*.

TODO: I'd like to come back and make an optimized version that doesn't have to crawl the tree multiple times. Maybe getting torque/weight/balanced status at the same time?

3.33.8 Question D: Implementation shakeup

Suppose we change the representation of mobiles so that the constructors are

```
1    (define (make-mobile left right)
2      (cons left right))
3
4    (define (make-branch length structure)
5      (cons length structure))
```

How much do you need to change your programs to convert to the new representation?

3.33.9 Answer D

Ideally I should only need to change the selectors, like this:

```
1    <<mobile-constructors-cons>>
2    (define (left-branch mobile)
3      (car mobile))
4    (define (right-branch mobile)
5      (cdr mobile))
6    (define (branch-length branch)
7      (car branch))
8    (define (branch-structure branch)
9      (cdr branch))
```

Now, if I run the same code, I should get the same result:

145

```
1   <<echo>>
2   <<mobile-selectors-cons>>
3   <<mobile-total-weight-2>>
4   <<mobile-balanced-dumb>>
5   (define M1 ;; all segments unbalanced
6     (make-mobile
7      (make-branch 5 5) ;; torque 25 = 5*5
8      (make-branch 1 ;; torque 5 = (2 + 3) * 1
9                    (make-mobile (make-branch 2 2)
10                                 (make-branch 2 3)))))
11  (define M2 ;; all segments balanced as they are duplicates
12    (make-mobile
13     (make-branch 2 ;; torque 8 = 2*(2+2)
14                   (make-mobile
15                    (make-branch 2 2)
16                    (make-branch 2 2)))
17     (make-branch 2
18                   (make-mobile
19                    (make-branch 2 2)
20                    (make-branch 2 2)))))
21  (define M3 ;; equal torque, but one segment is unbalanced.
22    (make-mobile
23     (make-branch 2 ;; torque 12 = 2*(4+2)
24                   (make-mobile
25                    (make-branch 2 4) ;; torque 8 = 2*4
26                    (make-branch 4 2))) ;torque 8 = 4*2
27     (make-branch 3 ;; torque 12 = 3*(1+3)
28                   (make-mobile
29                    (make-branch 1 1) ;; torque 1
30                    (make-branch 1 3))))) ;; torque 3
31  (define M4 ;; equal torque
32    (make-mobile
33     (make-branch 2 ;; torque 12 = 2*(4+2)
34                   (make-mobile
35                    (make-branch 2 4) ;; torque 8 = 2*4
36                    (make-branch 4 2))) ;torque 8 = 4*2
37     (make-branch 3 ;; torque 12 = 3*(1+3)
38                   (make-mobile
39                    (make-branch 3 1) ;; torque 3
40                    (make-branch 1 3))))) ;; torque 3
41  (define (isbalanced? Name Status)
42    (define Success "is balanced!")
43    (define Failure "is not balanced!")
44    (if (eq? Status #t)
45        (echo Name Success)
46        (echo Name Failure)))
47  (isbalanced? "M1" (balanced? M1))
48  (isbalanced? "M2" (balanced? M2))
```

146

```
49  (isbalanced? "M3" (balanced? M3))
50  (isbalanced? "M4" (balanced? M4))
```

```
M1 is not balanced!
M2 is balanced!
M3 is not balanced!
M4 is balanced!
```

3.34   Exercise 2.30

3.34.1   Question

Define a procedure `square-tree` analogous to the `square-@/list` procedure of
??. That is, `square-tree` should behave as follows:

```
1  (square-tree
2   (list 1
3         (list 2 (list 3 4) 5)
4         (list 6 7)))
5  ;; (1 (4 (9 16) 25) (36 49))
```

Define `square-tree` both directly (i.e., without using any higher-order pro-
cedures) and also by using `map` and recursion.

3.34.2   Answer

```
1  <<square>>
2  (define (square-tree-discrete tree)
3    (cond ((null? tree) '())
4          ((not (pair? tree)) (square tree))
5          (else (cons (square-tree-discrete (car tree))
6                      (square-tree-discrete (cdr tree))))))
7
8  (define (square-tree-map tree)
9    (map (lambda (sub-tree)
10           (if (pair? sub-tree)
11               (square-tree-map sub-tree)
12               (square sub-tree)))
13         tree))
```

```
1  (load "mattcheck2.scm")
2  <<square-tree>>
3  (let ((testlist
4         (list 1
5               (list 2 (list 3 4) 5)
6               (list 6 7)))
7        (answer
```

147

```
 8          (list 1
 9                 (list 4 (list 9 16) 25)
10                 (list 36 49))))
11     (mattcheck "square-tree-discrete"
12                (square-tree-discrete testlist)
13                answer)
14     (mattcheck "square-tree-map"
15                (square-tree-map testlist)
16                answer))
```

```
SUCCEED at square-tree-discrete
SUCCEED at square-tree-map
```

While writing that, I ran headfirst into a lesson I've had to repeatedly learn: default Guile functions end their lists with '() which does not match equality with lists ended with #nil.

```
 1   (let ((parens-list (cons 1 (cons 2 (cons 3 '()))))
 2         (nil-list (cons 1 (cons 2 (cons 3 #nil)))))
 3     (display parens-list)(display " ⟸ ends with '()")
 4     (newline)
 5     (display nil-list)(display " ⟸ ends with #nil")
 6     (newline)
 7     (display "Are these two lists equal? > ")
 8     (display (equal? parens-list nil-list))
 9     (newline)
10     (display "Does Guile consider #nil and '() equal? > ")
11     (display (equal? #nil '()))
12     (newline)
13     (display "What about #nil and #f? > ")
14     (display (equal? #nil #f)))
```

```
(1 2 3) ⟸ ends with '()
(1 2 3) ⟸ ends with #nil
Are these two lists equal? > #f
Does Guile consider #nil and '() equal? > #f
What about #nil and #f? > #f
```

3.35 Exercise 2.31

3.35.1 Question

Abstract your answer to **??** to produce a procedure `tree-map` with the property that `square-tree` could be defined as

```
 1   (define (square-tree tree) (tree-map square tree))
```

148

3.35.2 Answer

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

```
(load "mattcheck2.scm")
<<tree-map>>
<<square>>
(define (square-tree-tm tree)
  (tree-map square tree))
(let ((testlist
       (list 1
             (list 2 (list 3 4) 5)
             (list 6 7)))
      (answer
       (list 1
             (list 4 (list 9 16) 25)
             (list 36 49))))
  (mattcheck "square-tree-tm"
             (square-tree-tm testlist)
             answer))
```

```
SUCCEED at square-tree-tm
```

3.36 Exercise 2.32

3.36.1 Question

We can represent a set as a list of distinct elements, and we can represent the set of all subsets of the set as a list of lists. For example, if the set is (**1 2 3**), then the set of all subsets is ((**) (3) (2) (2 3) (1) (1 3) (1 2) (1 2 3)**). Complete the following definition of a procedure that generates the set of subsets of a set and give a clear explanation of why it works:

```
(define (subsets s)
  (if (null? s)
      (list nil)
      (let ((rest (subsets (cdr s))))
        (append rest (map <??> rest)))))
```

3.36.2 Answer

149

```scheme
1  (define (subsets s)
2    (if (null? s)
3        (list '())
4        (let ((rest (subsets (cdr s))))
5          (append rest (map (lambda(x)
6                              (cons (car s) x))
7                            rest)))))
```

```scheme
1   <<subsets>>
2   (load "mattcheck2.scm")
3   (let ((answer
4          (list '()
5                (list 3)
6                (list 2) (list 2 3)
7                (list 1) (list 1 3)
8                (list 1 2) (list 1 2 3))))
9     (mattcheck "subsets"
10               (subsets (list 1 2 3))
11               answer))
```

SUCCEED at subsets

Essentially, subsets is rotating through members of the list in a similar way that a counter incrementing rotates through all numbers in its base. For a list with items 1 to n, subsets makes a list with the last item, $[n]$, then lists $[n-1]$ and $[n-1, n]$, then lists $[n-2][n-2, n-1][n-2, n]$, then $[n-3][n-3, n-2][n-3, n-1][n-3, n]$ and so on.

I'd like to try adding some debugging statements to subsets and see if it might help clarify the operation.

```scheme
1   <<echo>>
2   (define (doit f x)
3     (f x) ;; this probably has a formal lambda calculus name
4     x)    ;; but I don't know what it is
5   (define (echo-return x)
6     (doit echo x))
7   (define (echo-y-return-x x . y)
8     (apply echo y)
9     x)
```

```scheme
1   <<subsets>>
2   <<echo-return>>
3   (define (subsets-debug s)
4     (echo "Enter with" s "{")
5     (let ((a1
6            (if (null? s)
7                (echo-return (list '()))
```

```
 8                  (let ((rest (subsets-debug (cdr s))))
 9                    (echo "[ iter" (car s) "over" rest "]")
10                    (append rest (map (lambda(x)
11                                        (let ((y (cons (car s) x)))
12                                          (format #t "~a " y)
13                                          y))
14                                      rest))))))
15       (echo "} end" s)
16       a1))
17   (let ((answer
18           (list '()
19                 (list 3)
20                 (list 2) (list 2 3)
21                 (list 1) (list 1 3)
22                 (list 1 2) (list 1 2 3))))
23     (subsets-debug (list 1 2 3))
24     answer)
```

```
Enter with (1 2 3) {
Enter with (2 3) {
Enter with (3) {
Enter with () {
(())
} end ()
[ iter 3 over (()) ]
(3)
} end (3)
[ iter 2 over (() (3)) ]
(2) (2 3)
} end (2 3)
[ iter 1 over (() (3) (2) (2 3)) ]
(1) (1 3) (1 2) (1 2 3)
} end (1 2 3)
```

3.37   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:

151
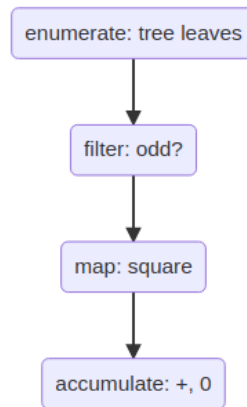
3.38 Exercise 2.33: The flexibility of `accumulate`

3.38.1 Text Definitions

```
1  (define (accumulate op initial sequence)
2    (if (null? sequence)
3        initial
4        (op (car sequence)
5            (accumulate op
6                        initial
7                        (cdr sequence)))))
```

3.38.2 Question

Fill in the missing expressions to complete the following definitions
of some basic list-manipulation operations as accumulations.

```
1  (define (map p sequence)
2    (accumulate (lambda (x y) <??>) nil sequence))
3  (define (append seq1 seq2)
4    (accumulate cons <??> <??>))
5  (define (length sequence)
6    (accumulate <??> 0 sequence))
```

3.38.3 Answer

```
1  (define (map-acc p sequence)
2    (accumulate (lambda (x y)
3                  (cons (p x) y))
4                '() sequence))
5  (define (append-acc seq1 seq2)
```

152

```
6        (accumulate cons seq2 seq1))
7     (define (length-acc sequence)
8       (accumulate (lambda(x y)
9                       (1+ y))
10                    0 sequence))
```

```
1    (load "mattcheck2.scm")
2    <<accumulate>>
3    <<accumulate-forms>>
4    <<square>>
5    (let ((l (list 1 2 3 4)))
6       (mattcheck "map"
7                   (map square l)
8                   (map-acc square l))
9       (mattcheck "append"
10                  (append l l)
11                  (append-acc l l))
12      (mattcheck "length"
13                  (length l)
14                  (length-acc l)))
```

```
SUCCEED at map
SUCCEED at append
SUCCEED at length
```

3.39   Exercise 2.34

3.39.1   Question

Evaluating a polynomial in x at a given value of x can be formulated as an accumulation. We evaluate the polynomial

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

using a well-known algorithm called *Horner's rule*, which structures the computation as

$$(\ldots (a_n x + a_{n-1}) x + \cdots + a_1) x + a_0.$$

In other words, we start with a_n, multiply by x, add a_{n-1}, multiply by x, and so on, until we reach a_0.

Fill in the following template to produce a procedure that evaluates a polynomial using Horner's rule. Assume that the coefficients of the polynomial are arranged in a sequence, from a_0 through a_n.

```
1   (define (horner-eval x coefficient-sequence)
2     (accumulate (lambda (this-coeff higher-terms) <??>)
3                 0
4                 coefficient-sequence))
```

For example, to compute $1 + 3x + 5x^3 + x^5$ at $x = 2$ you would evaluate

```
1   (horner-eval 2 (list 1 3 0 5 0 1))
```

3.39.2 Answer

```
1   (define (horner-eval x coefficient-sequence)
2     (accumulate (lambda (this-coeff higher-terms)
3                   (+ this-coeff
4                      (* higher-terms x)))
5                 0
6                 coefficient-sequence))
```

```
1   (load "mattcheck2.scm")
2   <<accumulate>>
3   <<horner-eval>>
4   (mattcheck "horner-eval"
5              (horner-eval 2 (list 1 3 0 5 0 1))
6              79)
```

```
SUCCEED at horner-eval
```

This one was very satisfying. It essentially "delays" the exponentiation, carrying it out per stage, by rewriting this:

$1 + 3 \times 2 + 5 \times 2^3 + 2^5$

Into this operation, left to right:

$$0 + 1 * 2 + 0 * 2 + 0 * 2 + 3 * 2 + 1$$

3.40   Exercise 2.35

3.40.1 Question

Redefine count-leaves from 2.2.2 as an accumulation:

3.40.2 Answer

```
1  (define (count-leaves-acc t)
2    (accumulate (lambda(i total)
3                  (+ i total))
4                0 (map (lambda(x)
5                         (if (pair? x)
6                             (count-leaves-acc x)
7                             1))
8                       t)))
```

```
1  (load "mattcheck2.scm")
2  <<accumulate>>
3  <<count-leaves>>
4  <<count-leaves-acc>>
5  (let ((l (list (list (list 1 2 3) 4) (list 5 (list 6 7) 8))))
6    (mattcheck "count-leaves-acc"
7              (count-leaves l)
8              (count-leaves-acc l)))
```

```
SUCCEED at count-leaves-acc
```

3.41   Exercise 2.36: Accumulate across multiple lists

3.41.1   Question

The procedure accumulate-n is similar to accumulate except that
it takes as its third argument a sequence of sequences, which are
all assumed to have the same number of elements. It applies the
designated accumulation procedure to combine all the first elements
of the sequences, all the second elements of the sequences, and so on,
and returns a sequence of the results. For instance, if s is a sequence
containing four sequences, ((1 2 3) (4 5 6) (7 8 9) (10 11 12)),
then the value of (accumulate-n + 0 s) should be the sequence (22
26 30). Fill in the missing expressions in the following definition of
accumulate-n:

```
1  (define (accumulate-n op init seqs)
2    (if (null? (car seqs))
3        nil
4        (cons (accumulate op init <??>)
5              (accumulate-n op init <??>))))
```

3.41.2   Answers

```
1  (define (accumulate-n op init seqs)
2    (if (null? (car seqs))
3        '()
```

155

```
4          (cons (accumulate op init
5                            (map car seqs))
6                (accumulate-n op init
7                              (map cdr seqs)))))
```

```
1  (load "mattcheck2.scm")
2  <<accumulate>>
3  <<accumulate-n>>
4  (let ((s (list (list 1 2 3) (list 4 5 6) (list 7 8 9) (list 10 11 12))))
5    (mattcheck "accumulate-n"
6               (accumulate-n + 0 s)
7               (list 22 26 30)))
```

```
SUCCEED at accumulate-n
```

3.42   Exercise 2.37: Enter the matrices

3.42.1   Question

See full quote in book.

Suppose we represent vectors as lists, and matrices as lists of vectors. For example:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 5 & 6 & 6 \\ 6 & 7 & 8 & 9 \end{pmatrix}$$

'((1 2 3 4) (4 5 6 6) (6 7 8 9))

Define these operations:

```
1  (define (dot-product v w)
2    (accumulate + 0 (map * v w)))
3  (define (matrix-*-vector m v)
4    (map <??> m))
5  (define (transpose mat)
6    (accumulate-n <??> <??> mat))
7  (define (matrix-*-matrix m n)
8    (let ((cols (transpose n)))
9      (map <??> m)))
```

3.42.2   Answer

```
1  (define (dot-product v w)
2    (accumulate + 0 (map * v w)))
3  (define (matrix-*-vector m v)
4    (map (lambda(row)
5            (dot-product v row))
6         m))
7  (define (transpose mat)
```

156

```
8      (accumulate-n cons '() mat))
9    (define (matrix-*-matrix m n)
10     (let ((cols (transpose n)))
11       (map (lambda(row)
12              (map (lambda(col)
13                     (dot-product row col))
14                   cols))
15            m)))
```

```
1    (load "mattcheck2.scm")
2    <<echo>>
3    <<accumulate>>
4    <<accumulate-n>>
5    <<matrix-ops>>
6    (let* ((v1 (list 1 2 3 4))
7           (v2 (list 5 6 7 8))
8           (m1 (list v1 v2))
9           (m1t (list (list 1 5)
10                      (list 2 6)
11                      (list 3 7)
12                      (list 4 8)))
13          (m1-*-m1t (list (list 30 70)
14                          (list 70 174))))
15     (mattcheck "transpose"
16                (transpose m1)
17                m1t)
18     (mattcheck "dot-product"
19                (dot-product v1 v2)
20                70)
21     (mattcheck "matrix-*-vector"
22                (matrix-*-vector m1 v1)
23                (list 30 70))
24     (mattcheck "matrix-*-matrix"
25                (matrix-*-matrix m1 m1t)
26                m1-*-m1t))
```

```
SUCCEED at transpose
SUCCEED at dot-product
SUCCEED at matrix-*-vector
SUCCEED at matrix-*-matrix
```

I struggled a lot with what order things should be processed and applied in. Some of that came from never having done matrix multiplication before now. I would probably still not understand it if I hadn't found Herb Gross' lecture regarding matrix operations[2].

The other issue is nested map operations. I find it easy to read Python-ish code like this:

---

[2] https://youtu.be/MfN1lqArwAg

```
1   # Pseudocode
2   for row in m1:
3       for column in m2t:
4           for a,b in row,column:
5               answer[i:j] += a*b
```

But much harder to comprehend Lisp code like this:

```
1   (map (lambda(row)
2         (map (lambda(col)
3               (dot-product row col))
4              m1t))
5        m1)
```

I must have a mental block in the way I think about map operations.

3.43   Exercise 2.38: fold-right

3.43.1   Question A

The `accumulate` procedure is also known as `fold-right`, because it combines the first element of the sequence with the result of combining all the elements to the right. There is also a `fold-left`, which is similar to `fold-right`, except that it combines elements working in the opposite direction:

```
1   (define (fold-left op initial sequence)
2     (define (iter result rest)
3       (if (null? rest)
4           result
5           (iter (op result (car rest))
6                 (cdr rest))))
7     (iter initial sequence))
```

What are the values of

```
1   (fold-right / 1 (list 1 2 3))
2   (fold-left / 1 (list 1 2 3))
3   (fold-right list nil (list 1 2 3))
4   (fold-left list nil (list 1 2 3))
```

3.43.2   Answer A

```
1   <<accumulate>>
2   (define fold-right accumulate)
3   <<fold-left>>
4   <<echo>>
```

```
5   (echo "(fold-right / 1 (list 1 2 3)):" (fold-right / 1 (list 1 2 3)))
6   (echo "(fold-left / 1 (list 1 2 3)):" (fold-left / 1 (list 1 2 3)))
7   (echo "(fold-right list nil (list 1 2 3))" (fold-right list '() (list 1
    ↪  2 3)))
8   (echo "(fold-left list nil (list 1 2 3))" (fold-left list '() (list 1 2
    ↪  3)))
```

```
(fold-right / 1 (list 1 2 3)): 3/2
(fold-left / 1 (list 1 2 3)): 1/6
(fold-right list nil (list 1 2 3)) (1 (2 (3 ())))
(fold-left list nil (list 1 2 3)) (((() 1) 2) 3)
```

3.43.3 Question B

Give a property that `op` should satisfy to guarantee that `fold-right` and `fold-left` will produce the same values for any sequence.

3.43.4 Answer B

They would need to be commutative, like addition and multiplication.

```
1   (load "mattcheck2.scm")
2   <<accumulate>>
3   (define fold-right accumulate)
4   <<fold-left>>
5   <<echo>>
6
7   (let* ((l (list 1 2 3 4 5))
8          (lr (reverse l)))
9     (mattcheck "commutative addition"
10              (fold-right + 0 l)
11              (fold-right + 0 lr)
12              (fold-left + 0 l)
13              (fold-left + 0 lr))
14    (mattcheck "commutative multiplication"
15              (fold-right * 1 l)
16              (fold-right * 1 lr)
17              (fold-left * 1 l)
18              (fold-left * 1 lr)))
```

```
SUCCEED at commutative addition
SUCCEED at commutative multiplication
```

3.44 Exercise 2.39: reverse via fold

3.44.1 Question

Complete the following definitions of `reverse` (Exercise 2.18) in terms of `fold-right` and `fold-left` from Exercise 2.38:

159

```
1  (define (reverse sequence)
2    (fold-right (lambda (x y) <??>) nil sequence))
3  (define (reverse sequence)
4    (fold-left (lambda (x y) <??>) nil sequence))
```

3.44.2   Answer

First, I'd like to start using the SRFI folds instead. This is my little "compatibility module".

```
1  ;; SICP compat
2  (use-srfis '(1))
3  (define accumulate fold-right)
4  (define fold-left fold)
```

Now to the problem.

```
1  <<fold-compat>>
2  (define (reverse-fr sequence)
3    (fold-right (lambda (x y)
4                  (append  y (list x)))
5                '() sequence))
6  (define (reverse-fl sequence)
7    (fold-left (lambda (x y)
8                  (append (list x) y))
9                '() sequence))
```

```
1   (load "mattcheck2.scm")
2   <<reverse-fold>>
3   (let ((l (iota 5))
4         (lr (reverse (iota 5))))
5     (mattcheck "reverse-fr"
6                (reverse-fr l)
7                lr)
8     (mattcheck "reverse-fl"
9                (reverse-fl l)
10               lr))
```

```
SUCCEED at reverse-fr
SUCCEED at reverse-fl
```

3.45   Exercise 2.40: `unique-pairs`

3.45.1   Text Definitions

```
1   (define (enumerate-interval low high)
2     (if (> low high)
3         '()
4         (cons low
5               (enumerate-interval
6                (+ low 1)
7                high))))
```

```
1   <<fold-compat>>
2   (define (flatmap proc seq)
3     (accumulate append '() (map proc seq)))
```

```
1   <<flatmap>>
2   <<enumerate-interval>>
3   <<prime-smallest-divisor>>
4   (define (prime-sum? pair)
5     (prime? (+ (car pair) (cadr pair))))
6   (define (make-pair-sum pair)
7     (list (car pair) (cadr pair) (+ (car pair) (cadr pair))))
8   (define (prime-sum-pairs n)
9     (map make-pair-sum
10         (filter prime-sum? (flatmap
11                             (lambda (i)
12                               (map (lambda (j) (list i j))
13                                    (enumerate-interval 1 (- i 1))))
14                             (enumerate-interval 1 n)))))
```

3.45.2   Question

Define a procedure `unique-pairs` that, given an integer n, generates
the sequence of pairs (i, j) with $1 \leq j < i \leq n$. Use `unique-pairs`
to simplify the definition of `prime-sum-pairs` given above.

3.45.3   Answer

```
1   <<fold-compat>>
2   <<enumerate-interval>>
3   <<flatmap>>
4   (define (unique-pairs n)
5     (flatmap
6      (lambda (i)
7        (map (lambda (j) (list i j))
8             (enumerate-interval 1 (- i 1))))
9      (enumerate-interval 1 n)))
```

161

```
1   (load "mattcheck2.scm")
2   <<unique-pairs>>
3   (let ((target 5)
4         (answer (list (list 2 1)(list 3 1)(list 3 2)(list 4 1)
5                       (list 4 2)(list 4 3)(list 5 1)(list 5 2)
6                       (list 5 3)(list 5 4))))
7     (mattcheck "unique-pairs"
8                (unique-pairs target)
9                answer))
10  <<echo>>
11  <<prime-sum-txt>>
12  (define (prime-sum-pairs-mine n)
13    (map make-pair-sum
14         (filter prime-sum?
15                 (unique-pairs n))))
16  (let ((answer (list (list 2 1 3) (list 3 2 5)
17                      (list 4 1 5) (list 4 3 7)
18                      (list 5 2 7)) ))
19    (mattcheck "prime-sum-pairs with unique-pairs"
20               (prime-sum-pairs-mine 5)
21               answer))
```

```
SUCCEED at unique-pairs
SUCCEED at prime-sum-pairs with unique-pairs
```

3.46   Exercise 2.41: Ordered triples of positive integers

3.46.1   Question

Write a procedure to find all ordered triples of distinct positive integers i, j, and k less than or equal to a given integer n that sum to a given integer s.

3.46.2   Answer

```
1   <<fold-compat>>
2   <<enumerate-interval>>
3   <<flatmap>>
4   (define (unique-triplets n)
5     (flatmap (lambda(i)
6                (flatmap (lambda(j)
7                           (map (lambda(k) (list i j k))
8                                (enumerate-interval 1 (- j 1))))
9                         (enumerate-interval 1 (- i 1))))
10             (enumerate-interval 1 n)))
```

```scheme
(define (triplets-sum n s)
  (filter (lambda(triplet)
            (= s (fold + 0 triplet)))
          (unique-triplets n)))
```

```scheme
(load "mattcheck2.scm")
<<unique-triplets>>
<<triplets-sum>>
(let ((answer-a
        (list (list 3 2 1) (list 4 2 1)
              (list 4 3 1) (list 4 3 2)
              (list 5 2 1) (list 5 3 1)
              (list 5 3 2) (list 5 4 1)
              (list 5 4 2) (list 5 4 3)))
      (answer-b
        (list (list 5 3 2) (list 5 4 1)
              (list 6 3 1) (list 7 2 1))))
  (mattcheck "unique-triplets"
             (unique-triplets 5)
             answer-a)
  (mattcheck "triplets-sum"
             (triplets-sum 7 10)
             answer-b))
```

```
SUCCEED at unique-triplets
SUCCEED at triplets-sum
```

3.47   Exercise 2.42: Eight Queens

3.47.1   Question

The "eight-queens puzzle" asks how to place eight queens on a chess-board so that no queen is in check from any other.

```scheme
<<flatmap>>
<<enumerate-interval>>
(define (queens board-size)
  (define (queen-cols k)
    (if (= k 0)
        (list empty-board)
        (filter
         (lambda (positions) (safe? k positions))
         (flatmap
          (lambda (rest-of-queens)
            (map (lambda (new-row)
                   (adjoin-position
                     new-row k rest-of-queens))
```

```
14                    (enumerate-interval 1 board-size)))
15              (queen-cols (- k 1))))))))
16        (queen-cols board-size))
```

Complete the program by writing the following:

- representation for sets of board positions, including:
  - adjoin-position, which adjoins a new row-column position to a set of positions
  - empty-board, which represents an empty set of positions.
- safe?, which determines for a set of positions, whether the queen in the kth column is safe with respect to the others. (Note that we need only check whether the new queen is safe— the other queens are already guaranteed safe with respect to each other.)

3.47.2   Answer

```
1    (define empty-board '())
2    (define (adjoin-position new-row column rest-of-queens)
3      (cons (list new-row column) rest-of-queens))
4    (define (newer-position board)
5      (car board))
6    (define (older-positions board)
7      (cdr board))
8    (define (get-row position)
9      (car position))
10   (define (get-column position)
11     (cadr position))
12   (define (safe? k board)
13     (define (same-row? a b)
14       (= (get-row a) (get-row b)))
15     (define (diagonal? a b)
16       (let ((row-diff (abs (- (get-row a) (get-row b))))
17             (col-diff (abs (- (get-column a) (get-column b)))))
18         (= row-diff col-diff)))
19     (let* ((new (newer-position board))
20            (compare (older-positions board)))
21       (and-map (lambda(pos)
22                   (not (or (same-row? new pos)
23                           (diagonal? new pos))))
24                compare)))
```

```
1    <<queens-txt>>
2    <<queens-mine>>
3    (load "mattcheck2.scm")
4    (let ((q4 '(((3 4) (1 3) (4 2) (2 1))
```

164

```
5              ((2 4) (4 3) (1 2) (3 1))))
6          (q11l 2680))
7      (mattcheck "queens"
8                 (list (queens 4)
9                       (length (queens 11)))
10                (list q4 q11l)))
```

SUCCEED at queens

3.48   Exercise 2.43: Louis' queens

3.48.1   Question

Louis Reasoner is having a terrible time doing Exercise 2.42. His q␣ueens procedure seems to work, but it runs extremely slowly. (Louis never does manage to wait long enough for it to solve even the 6×6 case.) When Louis asks Eva Lu Ator for help, she points out that he has interchanged the order of the nested mappings in the flatmap, writing it as

```
1    (flatmap
2     (lambda (new-row)
3       (map (lambda (rest-of-queens)
4              (adjoin-position new-row k rest-of-queens))
5            (queen-cols (- k 1))))
6     (enumerate-interval 1 board-size))
```

Explain why this interchange makes the program run slowly. Estimate how long it will take Louis's program to solve the eight-queens puzzle, assuming that the program in Exercise 2.42 solves the puzzle in time T.

3.48.2   Answer

The biggest contributor to the slowdown is likely the location of the queen-cols recursive call. This call being inside of the loop means it is being called k more times, all returning the same answer. But my math reasoning skills limit me from going further .Let's check with benchmarks.

```
1    <<queens-txt>>
2    <<queens-mine>>
3    (load "../mattbench.scm")
4
5    (define (queens-louis board-size)
6      (define (queen-cols k)
7        (if (= k 0)
8            (list empty-board)
9            (filter
```

```
10              (lambda (positions) (safe? k positions))
11             (flatmap
12              (lambda (new-row)
13                (map (lambda (rest-of-queens)
14                       (adjoin-position new-row k rest-of-queens))
15                     (queen-cols (- k 1))))
16              (enumerate-interval 1 board-size)))))
17      (queen-cols board-size))
18
19  (define (test size)
20     (format #t "~&normal queens x~a: ~a" size
21             (mattbench (lambda()(queens size)) 10000))
22     (format #t "~&swapped queens x~a: ~a" size
23             (mattbench (lambda()(queens-louis size)) 1000)))
24
25  (map (lambda(n)
26          (test n))
27        (enumerate-interval 5 8))
```
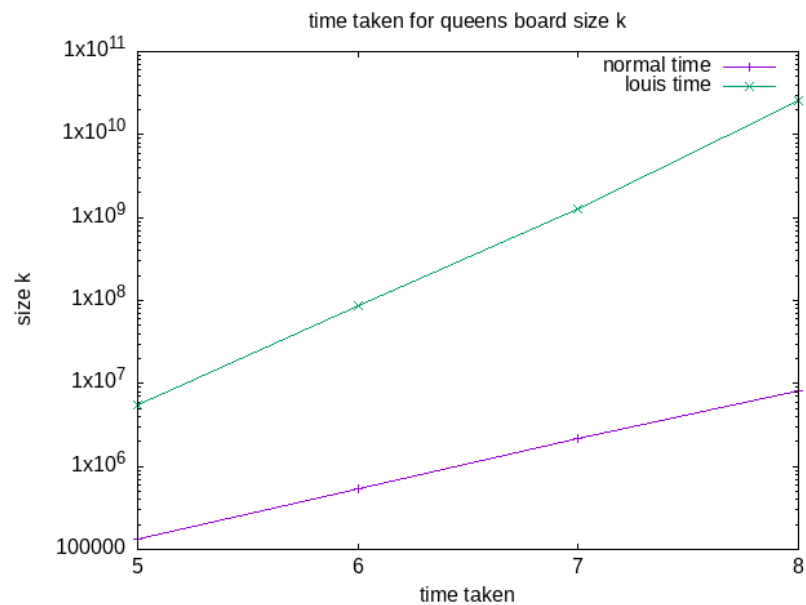
```
normal queens x5: 135424.6151
swapped queens x5: 5487381.643
normal queens x6: 538798.604
swapped queens x6: 85704466.218
normal queens x7: 2210394.5659
swapped queens x7: 1255288880.717
normal queens x8: 8067290.5992
swapped queens x8: 25384464494.259
```


time taken for queens board size k

So that's 40 times worse at 5x5, 159 times worse at 6x6, 568 times worse at 7x7, and 3146 times at 8x8.

after checking online with posts like this one [3] It looks like the big-O notation could be considered $\Theta((N^N) * T)$ at its simplest. I don't have a good grasp on how to reason out the time complexity of a non-trivial algorithm. I aim to eventually do "How to Solve It" and "How to Prove It" and possibly that will fill in the missing gaps before trying more serious comp-sci literature.

3.49   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:
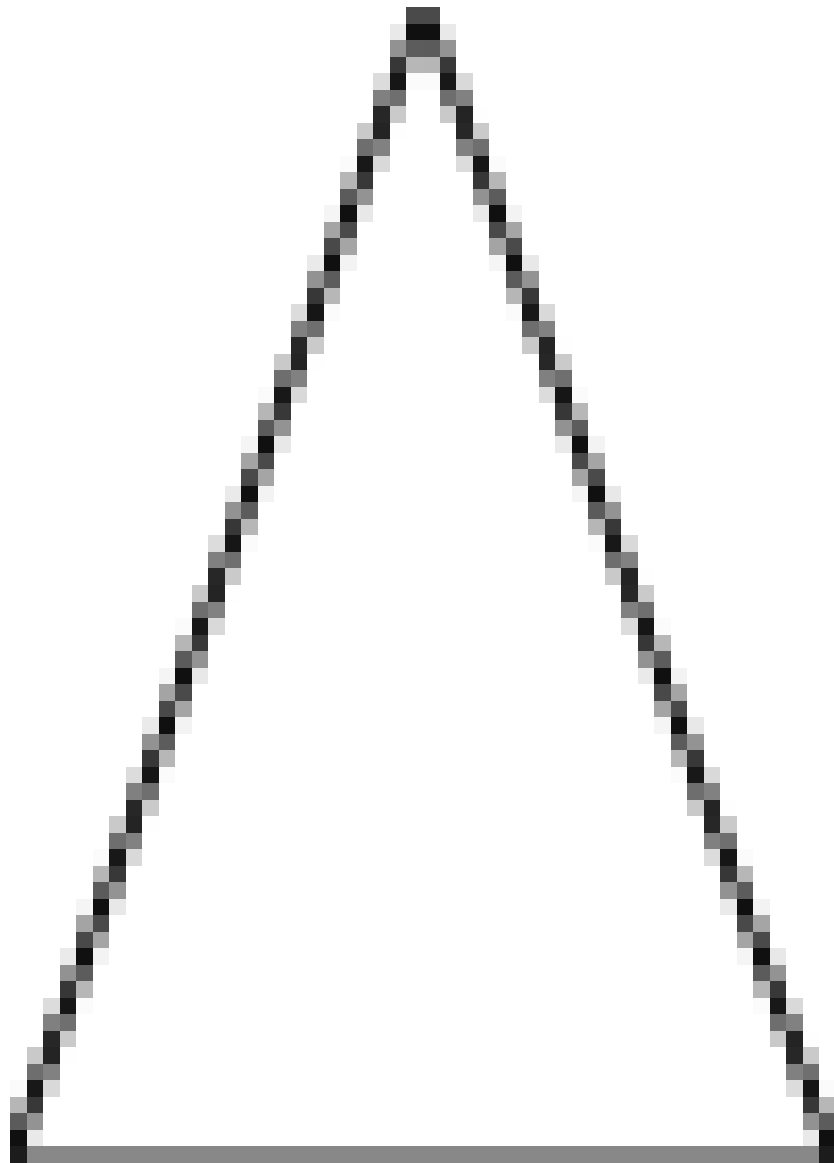
- a **painter** which makes an image within a specified parallelogram shaped frame. This is the most primitive element.

- **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.

I'm going to have to get a little messy in order to make the picture language a reality. First I need a support library called guile-picture-language[4] I installed Guix on my system and ran `guix shell guile-picture-language guile`. Now the `pict` module is availale to me.

```
(use-modules (pict))
(pict->file (triangle 50 70)
            "2/pict/test.svg")
```

---

[3] https://wernerdegroot.wordpress.com/2015/08/01/sicp-exercise-2-43/
[4] https://git.elephly.net/gitweb.cgi?p=software/guile-picture-language.git;a=blob;f=pict.scm;h=54f0ce09f4ac33c3ecc3340f038d4c9da5f4a443;hb=HEAD

So that works.
How about this?

```
1  (use-modules (pict))
2  (let* ((p (pict-from-file "2/pict/Potato.png"))
3         (pr (rotate p 180)))
4  (pict->file
5   (scale (ht-append (vl-append pr p) pr p (vc-append pr p))
```

```
6          0.5)
7    "2/pict/Rotato.svg")) ;; must be SVG
```


And I just realized there's no flip/mirror function in this library. Racket's picture language is more complete, but it won't integrate with org-mode how I need. Maybe I can mock up a text-based version, similar to the binary mobile, and simultaneously check my work with Racket.

```
1    ;; tilting at windmills trying to make a text-only picture language
2    ;; TODO, probably won't finish
3    (define pl-defaultsize 32)
4    (define (make-painter orientation height width)
5      (cons orientation (cons a b)))
6    (define (p-orientation p)
7      (car p))
8    (define (p-height p)
9      (cadr p))
10   (define (p-width p)
11     (cddr p))
12   (define (opposite orientation)
13     (cond ((eq? orientation 'down) 'up)
14           ((eq? orientation 'up ) 'down)
15           ((eq? orientation 'left) 'right)
16           ((eq? orientation 'right) 'left)))
17   (define (flip-p p)
18     (make-painter (opposite (p-orientation p))))
19   (define (below a b)
20     (make-painter 'down a b))
21   (define (beside a b)
22     (make-painter 'right a b))
23   (define (flip-vert p)
```

```
24    (if (pair? p)
25        (make-painter ))
26  (define (paint pict)
27    (define (rect p x y)
28      (cond ((string? p)
29              )))
30    (rec p pl-defaultsize pl-defaultsize))
31  (define (beside a b)
32    (hc-append a b))
33  (define (below a b)
34    (vc-append a b))
```

3.50   Exercise 2.44: `up-split`

3.50.1   Text Definitions

```
1  (define (right-split painter n)
2    (if (= n 0)
3        painter
4        (let ((smaller (right-split painter
5                                    (- n 1))))
6          (beside painter
7                  (below smaller smaller)))))
```

3.50.2   Question

Define the procedure `up-split` used by `corner-split`. It is similar to `right-split`, except that it switches the roles of `below` and `beside`.

3.50.3   Answer

```
1  (define (up-split painter n)
2    (if (= n 0)
3        painter
4        (let ((smaller (up-split painter
5                                 (- n 1))))
6          (below painter
7                  (beside smaller smaller)))))
```

3.51   Exercise 2.45: Generalized splitting

3.51.1   Question

`right-split` and `up-split` can be expressed as instances of a general splitting operation. Define a procedure `split` with the property that evaluating

```
1  (define right-split (split beside below))
2  (define up-split (split below beside))
```

produces procedures `right-split` and `up-split` with the same be-
haviors as the ones already defined.

3.51.2   Answer

```
1   (define (split f1 f2)
2     (define (rec painter n)
3       (if (= n 0)
4           painter
5           (let ((smaller (rec painter
6                                (- n 1))))
7             (f1 painter
8                 (f2 smaller smaller)))))
9     rec)
10  (define right-split (split beside below))
11  ;; This one goes down for some reason?
12  ;(define up-split (split below beside))
13  (define (up-split painter n)
14    (if (= n 0)
15        painter
16        (let ((smaller (up-split painter
17                                 (- n 1))))
18          (below painter
19                 (beside smaller smaller)))))
```

3.52   Exercise 2.46: Defining vectors

3.52.1   Question

A two-dimensional vector **v** running from the origin to a point can
be represented as a pair consisting of an x-coordinate and a y-
coordinate. Implement a data abstraction for vectors by giving a
constructor `make-vect` and corresponding selectors `xcor-vect` and
`ycor-vect`. In terms of your selectors and constructor, implement
procedures `add-vect`, `sub-vect`, and `scale-vect` that perform the
operations vector addition, vector subtraction, and multiplying a
vector by a scalar:

$$(x_1, y_1) + (x_2, y_2) = (x_1 + x_2, y_1 + y_2),$$
$$\[\quad (x_1, y_1) - (x_2, y_2) = (x_1 - x_2, y_1 - y_2), \quad \]$$
$$s \cdot (x, y) = (sx, sy).$$

3.52.2   Answer

This is pretty close to the `make-point` work done in Exercise 2.2, as well as my
2nd implementation of a rectangle in Exercise 2.3. Let's start there. Also, *aren't*

these points and not vectors since they don't have direction??

```scheme
;; Guarentee x is a float
(define (float x)
  (if (inexact? x)
      x
      (exact->inexact x)))
```

```scheme
<<force-float>>
(define (make-vect x y)
  (cons (float x)
        (float y))) ;; make-point
(define (xcor-vect v)
  (car v)) ;; x-point
(define (ycor-vect v)
  (cdr v)) ;; y-point
(define (add-vect v w)
  (make-vect (+ (xcor-vect v)
                (xcor-vect w))
             (+ (ycor-vect v)
                (ycor-vect w))))
(define (sub-vect v w)
  (make-vect (- (xcor-vect v)
                (xcor-vect w))
             (- (ycor-vect v)
                (ycor-vect w))))
(define (scale-vect s v)
  (make-vect (* s (xcor-vect v))
             (* s (ycor-vect v))))
```

I originally got the definition for `scale-vect` wrong by making it (`scale` v
s), then modified `frame-coord-map` when that broke. I realized my mistake once
I went to write `beside`.

3.53   Exercise 2.47: Defining frames

3.53.1   Question

Here are two possible constructors for frames:

```scheme
(define (make-frame origin edge1 edge2)
  (list origin edge1 edge2))
(define (make-frame origin edge1 edge2)
  (cons origin (cons edge1 edge2)))
```

For each constructor supply the appropriate selectors to produce an
implementation for frames.

172

3.53.2 Answer

```
1  (define (make-frame origin edge1 edge2)
2    (list origin edge1 edge2))
3  (define (origin-frame F)
4    (car F))
5  (define (edge1-frame F)
6    (cadr F))
7  (define (edge2-frame F)
8    (caddr F))
```

```
1  (define (make-frame origin edge1 edge2)
2    (cons origin (cons edge1 edge2)))
3  (define (origin-frame F)
4    (car F))
5  (define (edge1-frame F)
6    (cadr F))
7  (define (edge2-frame F)
8    (cddr F))
```

3.54 Exercise 2.48: Line segments

3.54.1 Question

A directed line segment in the plane can be represented as a pair of vectors—the vector running from the origin to the start-point of the segment, and the vector running from the origin to the end-point of the segment. Use your vector representation from Exercise 2.46 to define a representation for segments with a constructor `make-segm⌋ent` and selectors `start-segment` and `end-segment`.

3.54.2 Answer

Again reminding me of Exercise 2.2.

```
1  (define (make-segment start end)
2    (cons start end))
3  (define (start-segment seg)
4    (car seg))
5  (define (end-segment seg)
6    (cdr seg))
```

3.55 Exercise 2.49: Primitive painters

3.55.1 Text Definitions

```scheme
(define (frame-coord-map frame)
  (lambda (v)
    (add-vect
     (origin-frame frame)
     (add-vect
      (scale-vect (xcor-vect v)
                  (edge1-frame frame))
      (scale-vect (ycor-vect v)
                  (edge2-frame frame))))))
(define (segments->painter segment-list)
  (lambda (frame)
    (for-each
     (lambda (segment)
       (draw-line
        ((frame-coord-map frame)
         (start-segment segment))
        ((frame-coord-map frame)
         (end-segment segment))))
     segment-list)))
```

3.55.2 Question

Use `segments->painter` to define the following primitive painters:

1. The painter that draws the outline of the designated frame.
2. The painter that draws an X by connecting opposite corners of the frame.
3. The painter that draws a diamond shape by connecting the midpoints of the sides of the frame.
4. The `wave` painter.

3.55.3 Answer

In the past I would need to constantly execute my code to "see" what I'm doing. But I'm starting to think in larger chunks and need the feedback less.

```scheme
<<continuous-lines>>
(define outline
  (segments->painter (list
    (make-segment (make-vect 0 0)
                  (make-vect 0 1))
    (make-segment (make-vect 0 1)
                  (make-vect 1 1))
    (make-segment (make-vect 1 1)
                  (make-vect 1 0))
    (make-segment (make-vect 1 0)
                  (make-vect 0 0)))))
```

```
12  (define frame-X
13    (segments->painter (list
14     (make-segment (make-vect 0 0)
15                   (make-vect 1 1))
16     (make-segment (make-vect 0 1)
17                   (make-vect 1 0)))))
18  (define diamond
19    (segments->painter (list
20     (make-segment (make-vect 0.5 0)
21                   (make-vect 1 0.5))
22     (make-segment (make-vect 1 0.5)
23                   (make-vect 0.5 1))
24     (make-segment (make-vect 0.5 1)
25                   (make-vect 0 0.5))
26     (make-segment (make-vect 0 0.5)
27                   (make-vect 0.5 0)))))
```

I'll make the wave painter once I have some graph paper in front of me.

Ok, how can I verify this? I guess I could rewrite `segments->painter` for `pict`. Since this isn't a drawing system with imperative procedures, I should make it return a list of lines to be superimposed.

```
1   (use-modules (pict))
2   <<make-frame>>
3   <<make-vect>>
4   <<make-segment>>
5   (define (frame-coord-map frame)
6     ;; Returns a function for adjusting a frame by a vector
7     (lambda (v)
8       (add-vect
9        (origin-frame frame)
10       (add-vect
11        (scale-vect (xcor-vect v)
12                    (edge1-frame frame))
13        (scale-vect (ycor-vect v)
14                    (edge2-frame frame))))))
15  (define (draw-line start end)
16    ;; take two vectors, returns a line SVG object for pict
17    (line (xcor-vect start)
18          (ycor-vect start)
19          (xcor-vect end)
20          (ycor-vect end)))
21  (define (segments->painter segment-list)
22    ;; takes a list of segments, returns a "painter" lambda, which applies
          ↪  a frame
23    ;; to those segments and then maps over the result with draw-line to
          ↪  make a
24    ;; list of SVG line objects which pict can combine.
```

175

```
25    (lambda (frame)
26      (map
27       (lambda (segment)
28         (draw-line
29          ((frame-coord-map frame)
30           (start-segment segment))
31          ((frame-coord-map frame)
32           (end-segment segment))))
33       segment-list)))
34  ;; NOTE: in the text, draw-line is a function which triggers an action in
35  ;; some graphics driver, and returns nothing. Because of this, (map) was
36  ;; originally (for-each). Thus the final result would have been thrown
    ↪  away.
37
38  <<painters>>
39
40  (define (paint-lines painter)
41    ;; use pict to compile an SVG with the elements described by painter
42    (let ((Frame (make-frame (make-vect 0 0)
43                              (make-vect 500 0)
44                              (make-vect 0 500))))
45      (apply lt-superimpose
46             (painter Frame))))
```
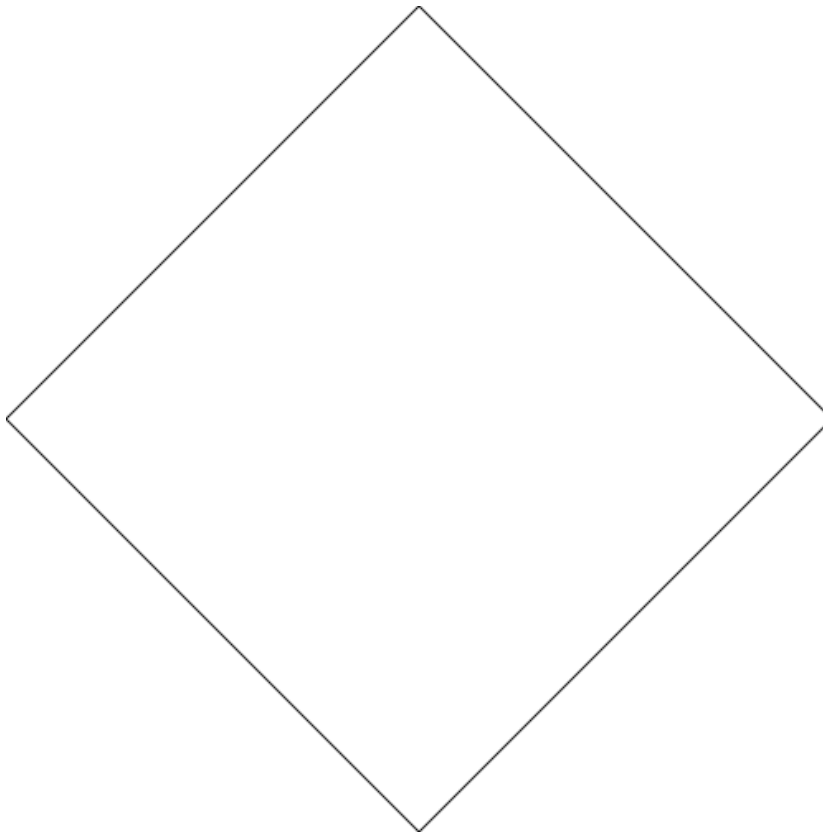
```
1   <<linepainter-pict>>
2   (pict->file (paint-lines diamond)
3               "2/pict/testline.svg")
4   ; #:maxw 100 #:maxh 100) <- FIXME: This procedure should take these
    ↪  arguments
5   ; but I can't get it to work.
```

Holy moly it actually works.

3.56 Exercise 2.50: Transforming painters

3.56.1 Text Definitions

```scheme
(define (transform-painter
          painter origin corner1 corner2)
  (lambda (frame)
    (let ((m (frame-coord-map frame)))
      (let ((new-origin (m origin)))
        (painter (make-frame new-origin
                             (sub-vect (m corner1)
                                       new-origin)
                             (sub-vect (m corner2)
                                       new-origin)))))))

(define (flip-vert painter)
  (transform-painter
```

```
14      painter
15      (make-vect 0.0 1.0)   ; new origin
16      (make-vect 1.0 1.0)   ; new end of edge1
17      (make-vect 0.0 0.0))) ; new end of edge2
18
19  (define (rotate90 painter)
20    (transform-painter painter
21                       (make-vect 1.0 0.0)
22                       (make-vect 1.0 1.0)
23                       (make-vect 0.0 0.0)))
24
25  (define (squash-inwards painter)
26    (transform-painter painter
27                       (make-vect 0.0 0.0)
28                       (make-vect 0.65 0.35)
29                       (make-vect 0.35 0.65)))
30  (define (beside painter1 painter2)
31    (let ((split-point (make-vect 0.5 0.0)))
32      (let ((paint-left  (transform-painter
33                            painter1
34                            (make-vect 0.0 0.0)
35                            split-point
36                            (make-vect 0.0 1.0)))
37            (paint-right (transform-painter
38                            painter2
39                            split-point
40                            (make-vect 1.0 0.0)
41                            (make-vect 0.5 1.0))))
42        (lambda (frame)
43          (append
44           (paint-left frame)
45           (paint-right frame))))))
```

```
1   <<linepainter-pict>>
2   <<frame-transforms-txt>>
3   (define topleft-tri
4     (segments->painter (list
5                          (make-segment (make-vect 0 0)
6                                        (make-vect 0 0.4))
7                          (make-segment (make-vect 0 0.4)
8                                        (make-vect 0.4 0))
9                          (make-segment (make-vect 0.4 0)
10                                        (make-vect 0 0)))))
11  (let ((picture (beside (beside topleft-tri
12                                 topleft-tri)
13                         (rotate90 topleft-tri))))
```

```
14   (pict->file (paint-lines picture)
15               "2/pict/rotatetest.svg"))
```


3.56.2   Question

Define the transformation `flip-horiz`, which flips painters horizontally, and transformations that rotate painters counterclockwise by
180 degrees and 270 degrees.

3.56.3   Answer

```
1    (define (flip-horiz painter)
2      (transform-painter
3       painter
4       (make-vect 1.0 0.0)   ; new origin
5       (make-vect 0.0 0.0)   ; new end of edge1
6       (make-vect 1.0 1.0))) ; new end of edge2
7
8    (define (rotate180 painter)
9      (transform-painter painter
10                         (make-vect 1.0 1.0)
11                         (make-vect 0.0 1.0)
12                         (make-vect 1.0 0.0)))
13   (define (rotate270 painter)
14     (transform-painter painter
15                         (make-vect 0.0 1.0)
16                         (make-vect 0.0 0.0)
17                         (make-vect 1.0 1.0)))
```

```
1    <<linepainter-pict>>
2    <<frame-transforms-txt>>
3    <<flip-rotate2>>
4    (define topleft-tri
5      (segments->painter (list
6                          (make-segment (make-vect 0 0)
```

179

```
7                                          (make-vect 0 0.4))
8                       (make-segment (make-vect 0 0.4)
9                                     (make-vect 0.8 0))
10                      (make-segment (make-vect 0.8 0)
11                                    (make-vect 0 0)))))
12  (let ((picture (beside (beside
13                          (flip-vert topleft-tri)
14                          (rotate90 topleft-tri))
15                         (beside
16                          (rotate180 topleft-tri)
17                          (rotate270 topleft-tri)))))
18     (pict->file (paint-lines picture)
19               "2/pict/rotate2.svg"))
```
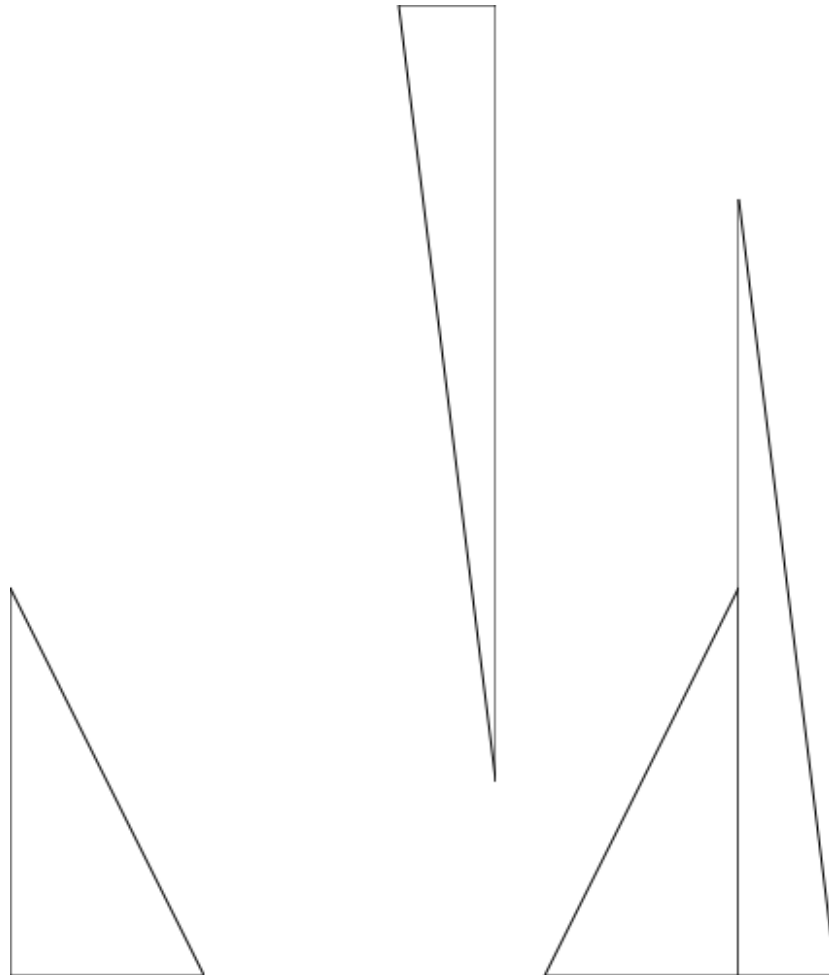
3.57 Exercise 2.51

3.57.1 Question

Define the `below` operation for painters. `below` takes two painters as arguments. The resulting painter, given a frame, draws with the first painter in the bottom of the frame and with the second painter in the top. Define `below` in two different ways—first by writing a procedure that is analogous to the `beside` procedure given above, and again in terms of `beside` and suitable rotation operations (from @ref{Exercise 2.50}).

3.57.2 Answer

```
(define (below painter1 painter2)
  (let ((split-point (make-vect 0.0 0.5)))
    (let ((paint-left  (transform-painter
                         painter1
                         (make-vect 0.0 0.0)
                         (make-vect 1.0 0.0)
                         split-point))
          (paint-right (transform-painter
                         painter2
                         split-point
                         (make-vect 1.0 0.5)
                         (make-vect 0.0 1.0))))
      (lambda (frame)
        (append
          (paint-left frame)
          (paint-right frame))))))
(define (below-rotate painter1 painter2)
  (rotate270 (beside
               (rotate90 painter2)
               (rotate90 painter1))))
```

```
<<linepainter-pict>>
<<frame-transforms-txt>>
<<flip-rotate2>>
<<below>>
(define topleft-tri
  (segments->painter (list
                       (make-segment (make-vect 0 0)
                                     (make-vect 0 0.4))
                       (make-segment (make-vect 0 0.4)
                                     (make-vect 0.8 0))
                       (make-segment (make-vect 0.8 0)
                                     (make-vect 0 0)))))
(let ((p1 (below (beside
```

```
14                              topleft-tri
15                              (rotate90 topleft-tri))
16                            (beside
17                              (rotate180 topleft-tri)
18                              (rotate270 topleft-tri))))
19        (p2 (below-rotate (beside
20                              topleft-tri
21                              (rotate90 topleft-tri))
22                            (beside
23                              (rotate180 topleft-tri)
24                              (rotate270 topleft-tri)))))
25      (pict->file (paint-lines (beside p1 p2))
26                  "2/pict/rotate3.svg"))
```
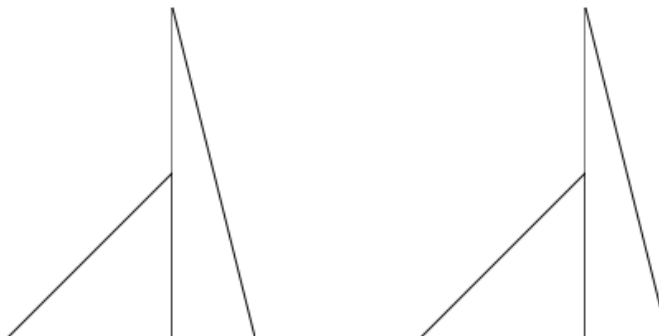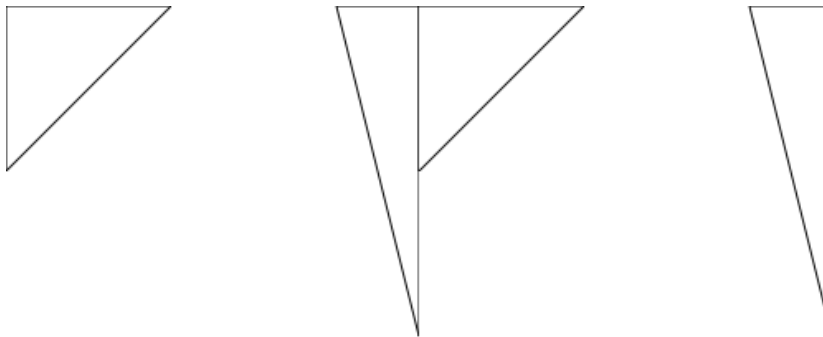

An aside: I'm beginning to see what makes Lisp-style programming different from C style. In C, the pictures would be described with separate data structures specified up front, but in Lisp you can use the code to create the data structure. Off the top of my head, the biggest players here would be first-level functions, and how statements evaluate to specific values rather than being imperative commands that cause something to happen elsewhere.

3.58   2.2.4 continued

stratified design is the notion that complex systems should be structured as a sequence of levels with a sequence of languages. See how electronic components are described in EE terms, the binary gates they form are described in digital logic terms, the programs they run described in programming language terms, the networks of programs described in network architecture terms, etc.

This stratified design can be seen in our picture language. We use lines and points to specify painters, use painters to make arrangements with beside/below, arrange these arrangements into higher-level arrangements like `up-split`.

3.59   Exercise 2.52

3.59.1   Question A

Make changes to the square limit of `wave` shown in Figure 2.9 by working at each of the levels described above. In particular:

- Add some segments to the primitive `wave` painter of Exercise 2.49 (to add a smile, for example).

3.59.2   Answer A

I need to do what I've been slacking off on: actually making the `wave` painter. First, I want a helper function to make inputting shapes easier.

```
(define (unwrap-if-needed args)
  ;; commonly needed in vararg functions
  (if (and (= 1 (length args))
           (list? (car args)))
      (car args) ; assume we were passed a list, unwrap it
      args))
```

```
<<unwrap-if-needed>>
(define (continuous-lines . vectors)
  ;; Given a list of vectors, return a series of line segments
  ;; that continuously follow the vectors until there are no more,
  ;; at which point connect the last vector to the first.
  (define (iter vecs lines)
    (if (> 2 (length vecs))
        lines
        (let* ((first (car vecs))
               (rest (cdr vecs))
               (second (car rest))
               (new-line (make-segment first second)))
          (iter rest (cons new-line lines)))))
```

```
14    (let ((input (unwrap-if-needed vectors)))
15      (iter (append input (list (car input))) ;; Connect to start
16            '())))
17  (define (numbers-to-vectors . numbers)
18    (define (iter input output)
19      (if (> 2 (length input))
20          output
21          (let* ((first (car input))
22                 (d (cdr input))
23                 (second (car d))
24                 (dd (cdr d)))
25            (iter dd
26                  (cons (make-vect first second)
27                        output)))))
28    (reverse (iter numbers '())))
```

```
1   <<linepainter-pict>>
2   <<frame-transforms-txt>>
3   <<flip-rotate2>>
4   <<below>>
5   <<continuous-lines>>
6   (define topleft-tri
7     (segments->painter (list
8                          (make-segment (make-vect 0 0)
9                                        (make-vect 0 0.4))
10                         (make-segment (make-vect 0 0.4)
11                                       (make-vect 0.8 0))
12                         (make-segment (make-vect 0.8 0)
13                                       (make-vect 0 0)))))
14  (define topleft-tri2
15    (segments->painter (continuous-lines
16                         (make-vect 0 0)
17                         (make-vect 0 0.4)
18                         (make-vect 0.8 0))))
19  (define topleft-tri3
20    (segments->painter (continuous-lines
21                         (numbers-to-vectors
22                          0 0
23                          0 0.4
24                          0.8 0))))
25  (let ((p1 (below (beside topleft-tri topleft-tri2)
26                   (beside topleft-tri topleft-tri3))))
27    (pict->file (paint-lines p1)
28                "2/pict/cl-test.svg"))
```

184
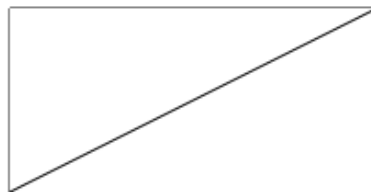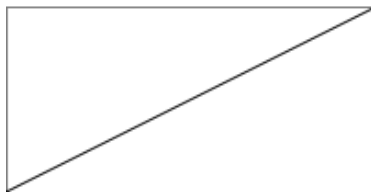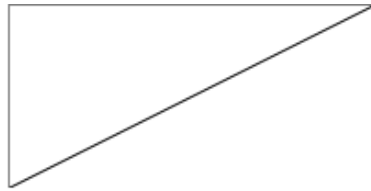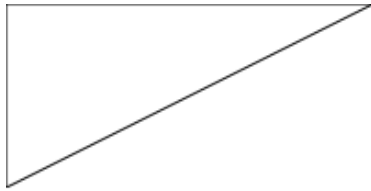
Now, let's write the wave painter.

```
<<continuous-lines>>
;; Should be raising their left hand and
;; lowering their right. (I screwed up the axes)
(define wave
  (segments->painter
   (continuous-lines
    (numbers-to-vectors
     0.55 0
     0.5 0.15 ;; center-left side of head
     0.55 0.3
     0.45 0.275
     0.25 0.35
     0 0.25
     0 0.3
     0.25 0.45
     0.45 0.35
     0.5 0.65
     0.4 1
     0.45 1
     0.6 0.7
     0.75 1
     0.8 1
     0.7 0.65
     0.75 0.35
```

```
25      1 0.675
26      1 0.6
27      0.75 0.275
28      0.65 0.3
29      0.7 0.15 ;; center-right side of head
30      0.65 0))))
```

```
1   <<linepainter-pict>>
2   <<frame-transforms-txt>>
3   <<flip-rotate2>>
4   <<below>>
5   <<continuous-lines>>
6   <<wave-painter>>
7   (let ((p1 (beside wave (flip-vert wave))))
8     (pict->file (paint-lines p1)
9                  "2/pict/wave-test.svg"))
```
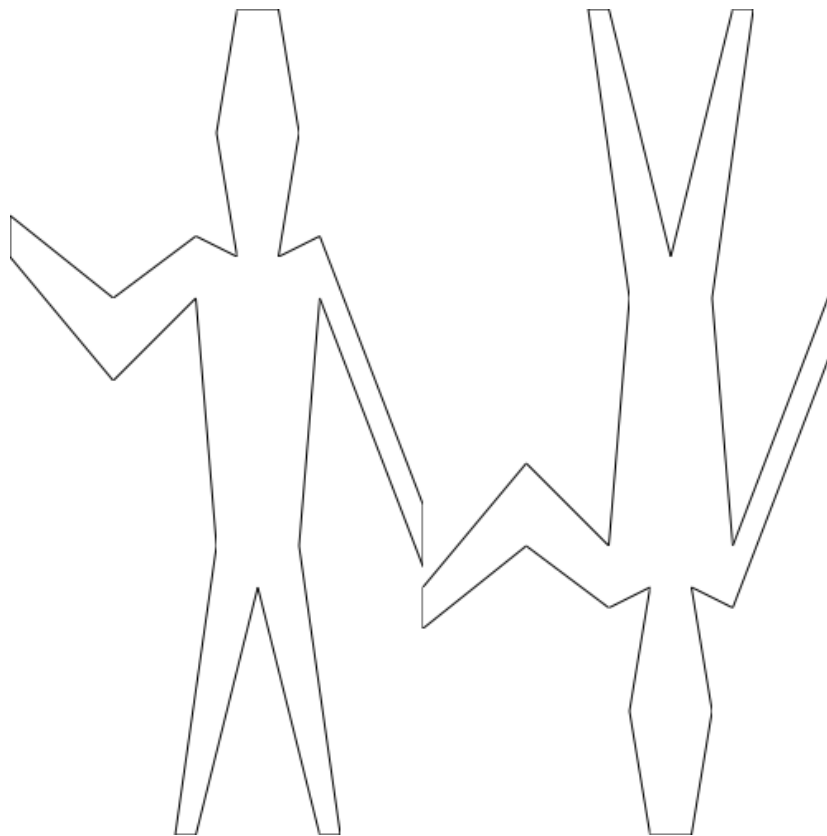
The question needs me to overlay something new. So I need some way to add more segments to a painter after it's already been written.

```
1   <<unwrap-if-needed>>
2   (define (append-painters-rec . args)
3     (lambda (frame)
4       (define (rec painters)
5         (if (null? painters)
6             '()
7             (append ((car painters) frame)
8                     (rec (cdr painters)))))
9       (rec (unwrap-if-needed args))))
10  (define (append-painters . args)
11    (lambda (frame)
12      (define (iter segments to-paint)
13        (if (null? to-paint)
14            segments
15            (iter (append ((car to-paint) frame)
16                          segments)
17                  (cdr to-paint))))
18      (iter '()
19            (unwrap-if-needed args))))
```

```
1   <<linepainter-pict>>
2   <<frame-transforms-txt>>
3   <<flip-rotate2>>
4   <<below>>
5   <<continuous-lines>>
6   <<painters>>
7   <<wave-painter>>
8   <<append-painters>>
9   (let ((p1 (append-painters
10             (beside wave (flip-vert wave))
11             frame-X
12             outline)))
13    (pict->file (paint-lines p1)
14                "2/pict/append-test.svg"))
```

Now let's (try to) add a smile.

```
<<wave-painter>>
<<append-painters>>
(define wave-smile
  (append-painters
   wave
   (segments->painter
    (continuous-lines
     (numbers-to-vectors
      0.55 0.2
      0.6 0.225
      0.65 0.2)))))
```

```
<<linepainter-pict>>
<<frame-transforms-txt>>
<<flip-rotate2>>
<<below>>
<<continuous-lines>>
```

```
6   <<wave-smile>>
7   (pict->file (paint-lines wave-smile)
8               "2/pict/wave-smile-test.svg")
```
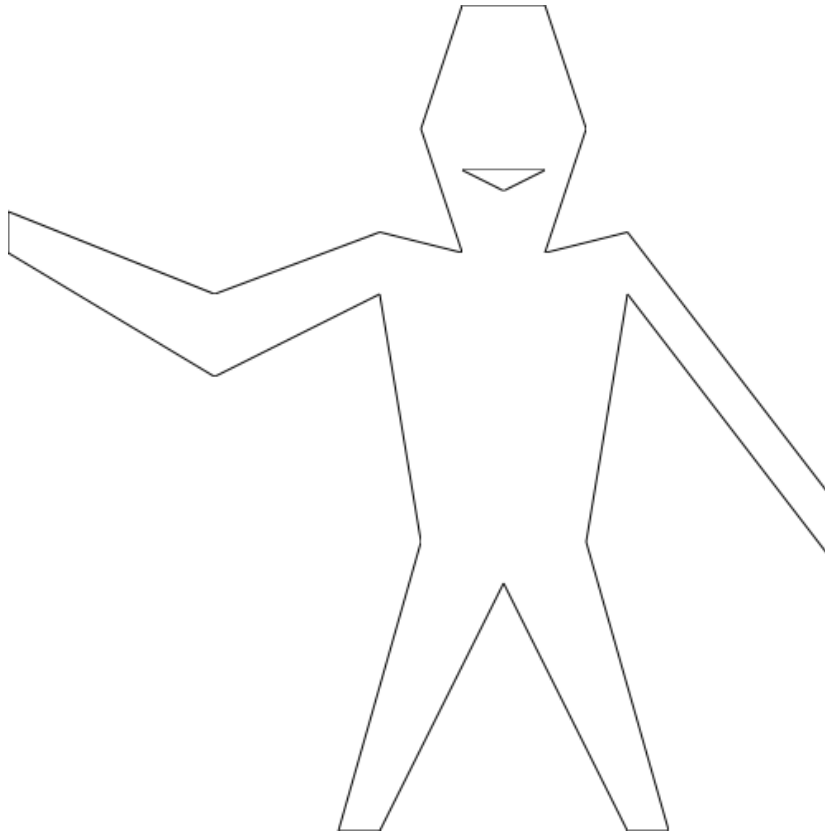

3.59.3   Question B

Change the pattern constructed by `corner-split` (for example, by using only one copy of the `up-split` and `right-split` images instead of two).

```
1    (define (corner-split painter n)
2      (if (= n 0)
3          painter
4          (let ((up (up-split painter (- n 1)))
5                (right (right-split painter
6                                    (- n 1))))
7            (let ((top-left (beside up up))
8                  (bottom-right (below right
9                                       right))
10                 (corner (corner-split painter
```

189

```
11                                            (- n 1))))
12            (beside (below painter top-left)
13                    (below bottom-right
14                           corner))))))

15

16  (define (corner-split-mine painter n)
17    (if (= n 0)
18        painter
19        (let ((up (up-split painter (- n 1)))
20              (right (right-split painter
21                                  (- n 1))))
22          (let ((top-left up)
23                (bottom-right right)
24                (corner (corner-split-mine painter
25                                           (- n 1))))
26            (beside (below painter top-left)
27                    (below bottom-right
28                           corner))))))
```
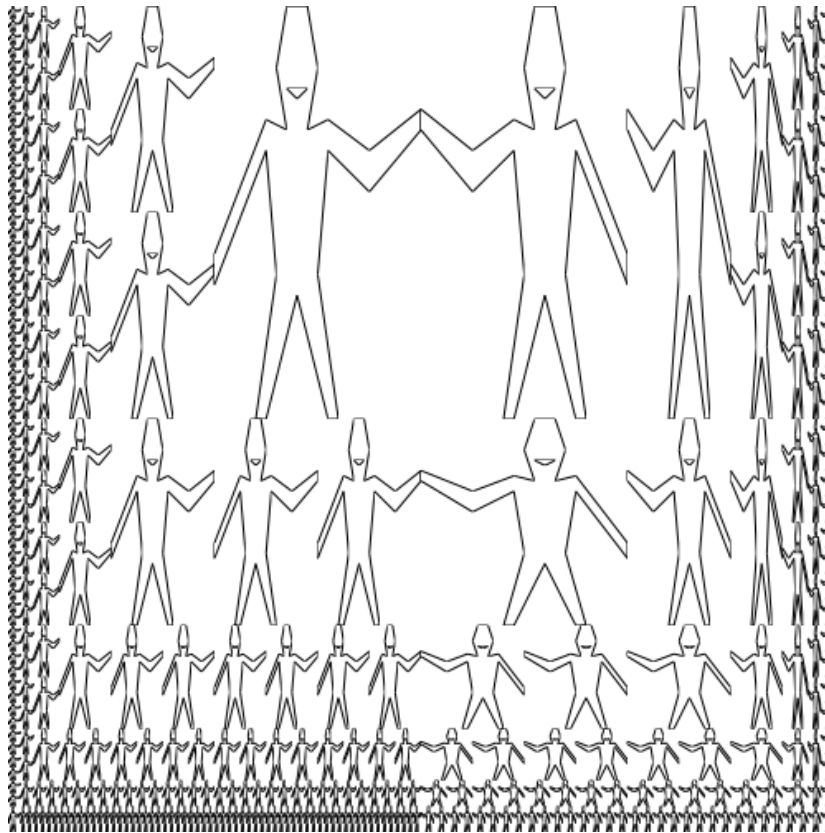
```
1   <<linepainter-pict>>
2   <<frame-transforms-txt>>
3   <<flip-rotate2>>
4   <<below>>
5   <<continuous-lines>>
6   <<painters>>
7   <<wave-smile>>
8   <<append-painters>>
9   <<splits-generalized>>
10  <<corner-split>>
11  (let ((p1 (beside (flip-horiz (corner-split wave-smile 5))
12                    (corner-split-mine wave-smile 5))))
13  ;(let ((p1 (up-split frame-X 5)))
14    (pict->file (paint-lines p1)
15                "2/pict/corner-splits.svg"))
```

So my modified version doesn't split at the `corner-split` level, as predicted. However `up-split` and `right-split` do, so the effect is only delayed by one level. But more importantly:

Why does it go down instead of up? I don't get it.

`corner-split` and `up-split` are even the same code as every answer online. Could my `paint-lines` procedure be what's causing issues?

3.59.4   Question C

Modify the version of `square-limit` that uses `square-of-four` so as to assemble the corners in a different pattern. (For example, you might make the big Mr. Rogers look outward from each corner of the square.)

3.59.5   Textbook Definitions

```
(define (square-of-four tl tr bl br)
  (lambda (painter)
    (let ((top (beside (tl painter)
```

```
 4                            (tr painter)))
 5              (bottom (beside (bl painter)
 6                              (br painter))))
 7          (below bottom top))))
 8   (define (square-limit painter n)
 9     (let ((combine4
10            (square-of-four flip-horiz
11                            identity
12                            rotate180
13                            flip-vert)))
14       (combine4 (corner-split painter n))))
```

3.59.6 Answer C

```
 1   (define (square-limit-mine painter n)
 2     (let ((combine4
 3            (square-of-four identity
 4                            flip-horiz
 5                            flip-vert
 6                            rotate180)))
 7       (combine4 (corner-split painter n))))
```

```
 1   <<linepainter-pict>>
 2   <<frame-transforms-txt>>
 3   <<flip-rotate2>>
 4   <<below>>
 5   <<continuous-lines>>
 6   <<wave-smile>>
 7   <<splits-generalized>>
 8   <<corner-split>>
 9   <<square-of-four-txt>>
10   <<square-limit-mine>>
11   (pict->file (paint-lines (beside (square-limit wave-smile 5)
12                                    (square-limit-mine wave-smile 5)))
13             "2/pict/square-limits.svg")
```

Ok, I'm confused. Shouldn't the shape of `square-limit` be the same regardless of the operators passed to it? It looks like the operators effect each chunk separately.

```
1    <<linepainter-pict>>
2    <<frame-transforms-txt>>
3    <<flip-rotate2>>
4    <<below>>
5    <<continuous-lines>>
6    <<wave-smile>>
7    <<splits-generalized>>
8    <<corner-split>>
9    <<square-of-four-txt>>
10   (define (square-limit-mine painter n)
11     (let ((combine4
12            (square-of-four identity
13                            identity
14                            identity
15                            identity)))
```
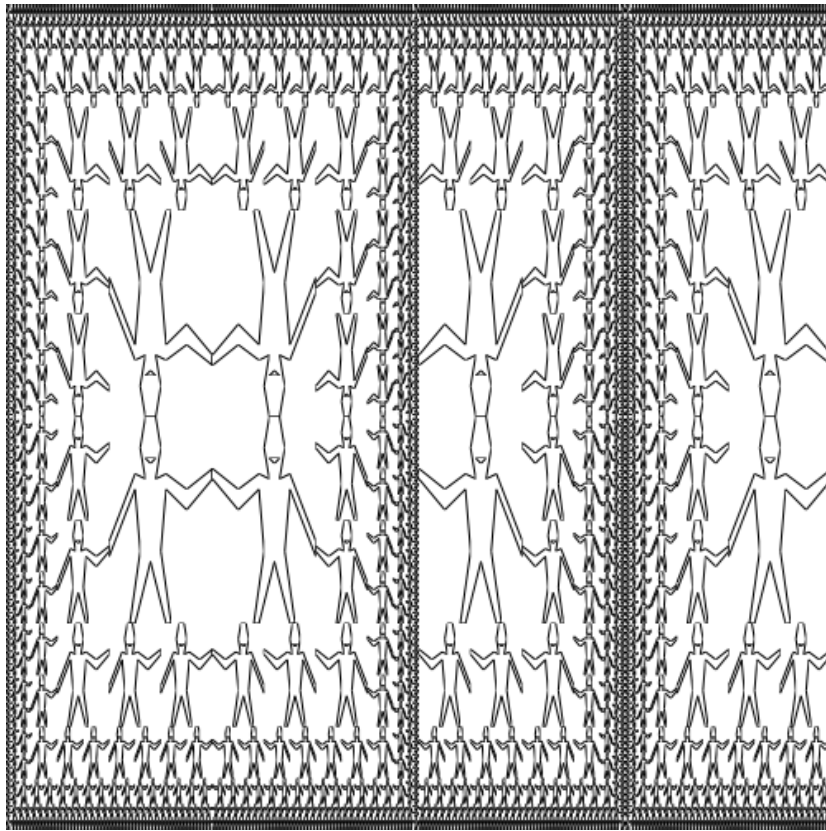
```
16          (combine4 (corner-split painter n))))
17    (pict->file (paint-lines (square-limit-mine wave-smile 5))
18                 "2/pict/square-limit-identity.svg")
```
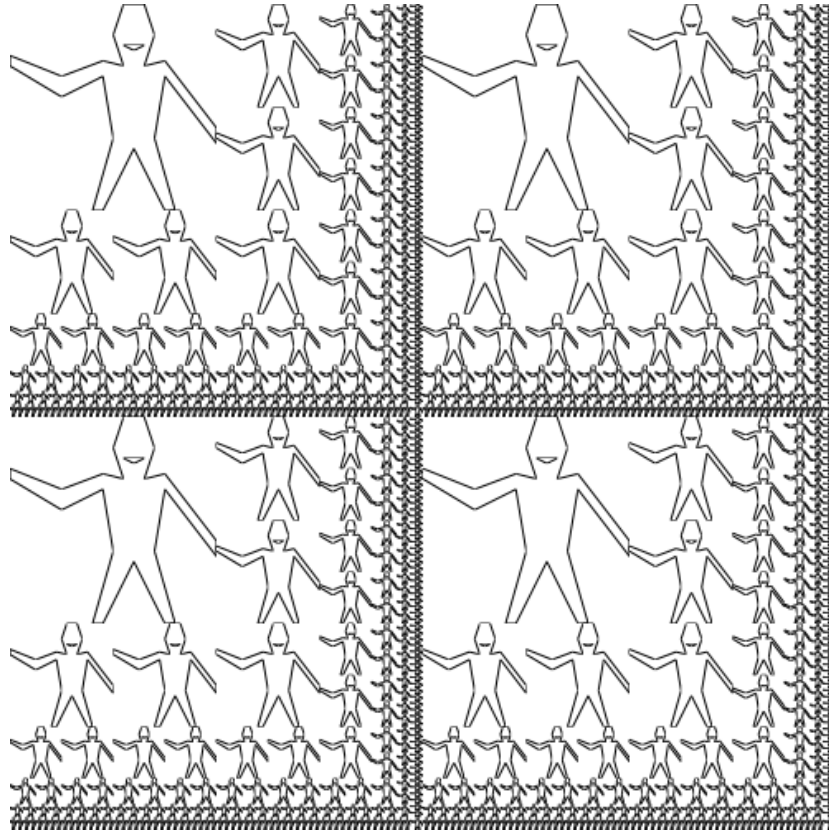

I expected the operators would change the *image orientation*, not the *structure*. Maybe this is because my `draw-lines` makes recursively nested stacks of painter objects, rather than imperative calls to drawing procedures? Looking on the internet I see someone else having the same results.[5]

3.60 2.3.1: Quotation

Syntactic vs semantic use in Lisp:

```
1    (define dog 123)
2    (+ dog dog) ;; semantic usage
3    (quote dog) ;; syntactic usage
```

This was introduced by famous LISP scientists, Abbott and Costello.

---

[5] http://pipecode.blogspot.com/2013/04/sicp-exercise-252.html

3.61 Exercise 2.53

3.61.1 Question

What would the interpreter print in response to evaluating each of the following expressions?

```
(list 'a 'b 'c)
(list (list 'george))
(cdr '((x1 x2) (y1 y2)))
(cadr '((x1 x2) (y1 y2)))
(pair? (car '(a short list)))
(memq 'red '((red shoes) (blue socks)))
(memq 'red '(red shoes blue socks))
```

3.61.2 Answer

```
(list 'a 'b 'c) ;; (a b c)
(list (list 'george)) ;; ((george))
(cdr '((x1 x2) (y1 y2))) ;; ((y1 y2))
(cadr '((x1 x2) (y1 y2))) ;; (y1 y2)
(pair? (car '(a short list))) ;; false
(memq 'red '((red shoes) (blue socks))) ;; false
(memq 'red '(red shoes blue socks)) ;; (red shoes blue socks)
```

3.62 Exercise 2.54

Two lists are said to be `equal?` if they contain equal elements arranged in the same order. For example,

```
(equal? '(this is a list) '(this is a list))
```

is true, but

```
(equal? '(this is a list) '(this (is a) list))
```

is false. To be more precise, we can define `equal?` recursively in terms of the basic `eq?` equality of symbols by saying that `a` and `b` are `equal?` if they are both symbols and the symbols are `eq?`, or if they are both lists such that (`car a`) is `equal?` to (`car b`) and (`cdr a`) is `equal?` to (`cdr b`). Using this idea, implement `equal?` as a procedure.

3.62.1 Answer

```
1  (define (equal? a b)
2    (cond ((and (symbol? a)
3                (symbol? b))
4           (eq? a b))
5          ((and (list? a)
6                (list? b))
7           (and (equal? (car a) (car b))
8                (equal? (cdr a) (cdr b))))
9          (else #f)))
```

```
1   (load "mattcheck2.scm")
2   (let ((a '(this is a list))
3         (aa '(this is a list))
4         (b '(this (quote is a) list)))
5     (mattcheck "equal? true"
6                (equal? a a))
7     (mattcheck "equal? trick question"
8                (equal? a a))
9     (mattcheck "equal? false"
10               (equal? a b)
11               #f))
```

```
SUCCEED at equal? true
SUCCEED at equal? trick question
SUCCEED at equal? false
```

3.63   Exercise 2.55

3.63.1   Question

Eva Lu Ator types to the interpreter the expression

```
1   (car ''abracadabra)
```

To her surprise, the interpreter prints back **quote**. Explain.

3.63.2   Answer

You're quoting quote, silly! Who's on first?

3.64   2.3.2: Example: Symbolic differentiator

I needed to get some help, but I think I have some understanding now.

The **derivative** of an expression E, relative to the variable x, describes the rate of change (or δ **delta**) of that expression with relation to x. Mathematicians symbolize it like this:

196

$$\frac{dE}{dx}$$

Two stumbling blocks about this notation:

1. This is not division. I understand why this notation would make sense to an experienced mathematician, since you factor out things by dividing an expression. For example:

$$\frac{3x}{x} = 3$$

$$\frac{3x}{3} = x$$

But it's still confusing.

2. dx is not some variable d times some variable x. $\frac{d(x+3)}{dx}$ means "the derivative of $x + 3$ with respect to x."

The process of finding the derivative is called **differentiation**.
The rules the book puts forward, rephrased in plain English, are these:

- If the expression is a constant, the derivative must be 0.

- If the expression is x, the derivative must be 1.

- If the expression is $u + v$, the derivative is the sum of two derivatives:

    1. The derivative of u with respect to x.
    2. The derivative of v with respect to x.

- If the expression is $u \times v$, the derivative is the sum of:

    1. u times the derivative of v with respect to x
    2. v times the derivative of u with respect to x

Later:

- If the expression is u^0, the result is 1, which is a constant, so the rate of change is 0.

- If the expression is u^1, the result is u, so the rate of change is 1.

- Else, if the expression is u^n, the derivative is the product of these:

    1. n
    2. u to the power of $n - 1$
    3. The derivative of u with respect to x.

3.65   Exercise 2.56: Differentiating exponentiation

3.65.1   Text definitions

```
(define (variable? x) (symbol? x))
(define (same-variable? v1 v2)
  (and (variable? v1) (variable? v2) (eq? v1 v2)))
(define (sum? x) (and (pair? x) (eq? (car x) '+)))
(define (addend s) (cadr s))
(define (augend s) (caddr s))
(define (product? x) (and (pair? x) (eq? (car x) '*)))
(define (multiplier p) (cadr p))
(define (multiplicand p) (caddr p))
(define (make-sum a1 a2)
  (cond ((=number? a1 0) a2)
        ((=number? a2 0) a1)
        ((and (number? a1) (number? a2))
         (+ a1 a2))
        (else (list '+ a1 a2))))
(define (=number? exp num)
  (and (number? exp) (= exp num)))
(define (make-product m1 m2)
  (cond ((or (=number? m1 0) (=number? m2 0)) 0)
        ((=number? m1 1) m2)
        ((=number? m2 1) m1)
        ((and (number? m1) (number? m2)) (* m1 m2))
        (else (list '* m1 m2))))
(define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp) (if (same-variable? exp var) 1 0))
        ((sum? exp) (make-sum (deriv (addend exp) var)
                              (deriv (augend exp) var)))
        ((product? exp)
         (make-sum
           (make-product (multiplier exp)
                         (deriv (multiplicand exp) var))
           (make-product (deriv (multiplier exp) var)
                         (multiplicand exp))))
        (else
         (error "unknown expression type: DERIV" exp))))
```

3.65.2   Question

Show how to extend the basic differentiator to handle more kinds of expressions. For instance, implement the differentiation rule

$$\frac{d\left(u^{n}\right)}{dx} = nu^{n-1}\frac{du}{dx}$$

198

by adding a new clause to the `deriv` program and defining appropriate procedures `exponentiation?`, `base`, `exponent`, and `make-expon⌋entiation`. (You may use the symbol `**` to denote exponentiation.) Build in the rules that anything raised to the power 0 is 1 and anything raised to the power 1 is the thing itself.

3.65.3   Answer

```scheme
(define (exponentiation? x) (and (pair? x) (eq? (car x) '**)))
(define (base e) (cadr e))
(define (exponent e) (caddr e))
(define (make-exponentiation b e)
  (cond ((=number? b 1) 1)
        ((=number? e 0) 1)
        ((=number? e 1) b)
        ((and (number? b) (number? e)) (expt b e))
        (else (list '** b e))))
(define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp) (if (same-variable? exp var) 1 0))
        ((sum? exp) (make-sum (deriv (addend exp) var)
                              (deriv (augend exp) var)))
        ((product? exp)
         (make-sum
          (make-product (multiplier exp)
                        (deriv (multiplicand exp) var))
          (make-product (deriv (multiplier exp) var)
                        (multiplicand exp))))
        ((exponentiation? exp)
         (make-product
          (make-product
           (exponent exp)
           (make-exponentiation (base exp)
                                (make-sum (exponent exp) -1)))
          (deriv (base exp) var)))
        (else
         (error "unknown expression type: DERIV" exp))))
```

First I'll make sure the textbook examples work as expected.

```scheme
<<deriv-basic-txt>>
<<deriv-exp>>
(load "mattcheck2.scm")
(mattcheck "deriv: basic addition"
           (deriv '(+ x 3) 'x)
           1)
(mattcheck "deriv: basic multiplication"
           (deriv '(* x y) 'x)
```

```
 9                'y)
10   (mattcheck "deriv: bad simplification"
11              (deriv '(* (* x y) (+ x 3)) 'x)
12              '(+ (* x y) (* y (+ x 3)))))
13   ;; Now, did I do my job right?
14   ;; Checking against this guy's results:
15   ;;   http://jots-jottings.blogspot.com/2011/11/sicp-exercise-256-differe⌋
     ↪  ntiating.html
16   (mattcheck "make-exponentiation 1"
17              (make-exponentiation 1 12)
18              1)
19   (mattcheck "make-exponentiation 2"
20              (make-exponentiation 2 12)
21              4096)
22   (mattcheck "make-exponentiation 3"
23              (make-exponentiation 'x 12)
24              '(** x 12))
25   (mattcheck "deriv: exponentiation 1"
26              (deriv (make-exponentiation 'a 5) 'a)
27              '(* 5 (** a 4)))
28   (mattcheck "deriv: exponentiation 2"
29              (deriv (make-exponentiation 'a 'b) 'a)
30              '(* b (** a (+ b -1))))
```

```
SUCCEED at deriv: basic addition
SUCCEED at deriv: basic multiplication
SUCCEED at deriv: bad simplification
SUCCEED at make-exponentiation 1
SUCCEED at make-exponentiation 2
SUCCEED at make-exponentiation 3
SUCCEED at deriv: exponentiation 1
SUCCEED at deriv: exponentiation 2
```

3.66   Exercise 2.57: Differentiating arbitrary-length expressions

3.66.1   Question

Extend the differentiation program to handle sums and products of arbitrary numbers of (two or more) terms. Then the last example above could be expressed as

```
1   (deriv '(* x y (+ x 3)) 'x)
```

Try to do this by changing only the representation for sums and products, without changing the deriv procedure at all. For example, the addend of a sum would be the first term, and the augend would be the sum of the rest of the terms.

3.66.2 Answer

```scheme
(define (sum? x) (and (pair? x) (eq? (car x) '+)))
(define (addend s) (cadr s))
(define (augend s)
  (let ((rest (cddr s)))
    (if (null? (cdr rest))
        (car rest)
;         (make-sum (addend rest)
;                   (augend (cons '+ rest))))))
        (cons '+ rest))))
(define (product? x) (and (pair? x) (eq? (car x) '*)))
(define (multiplier p) (cadr p))
(define (multiplicand p)
  (let ((rest (cddr p)))
    (if (null? (cdr rest))
        (car rest)
;         (make-product (multiplier rest)
;                       (multiplicand (cons '* rest))))))
        (cons '* rest))))
(define (make-sum a1 a2)
  (cond ((=number? a1 0) a2)
        ((=number? a2 0) a1)
        ((and (number? a1) (number? a2))
         (+ a1 a2))
        (else (list '+ a1 a2))))
(define (make-product m1 m2)
  (cond ((or (=number? m1 0) (=number? m2 0)) 0)
        ((=number? m1 1) m2)
        ((=number? m2 1) m1)
        ((and (number? m1) (number? m2))
         (* m1 m2))
        (else (list '* m1 m2))))
```

```scheme
<<deriv-basic-txt>>
<<deriv-exp>>
<<deriv-longer>>
(load "mattcheck2.scm")
(mattcheck "deriv: longer addition"
           (deriv '(+ x 3 6 z) 'x)
           1)
(mattcheck "deriv: longer multiplication"
           (deriv '(* x y (+ x 3)) 'x)
           '(+ (* x y) (* y (+ x 3))))
```

```
SUCCEED at deriv: longer addition
SUCCEED at deriv: longer multiplication
```

I had to look up the solution for this one. The commented sections in augend
and multiplicand were the answers I was trying to make work:

```
1   (define (augend s)
2     (let ((rest (cddr s)))
3       (if (null? (cdr rest))
4           (car rest)
5   ;         (make-sum (addend rest)
6   ;                   (augend (cons '+ rest))))))))
7           (cons '+ rest))))
```

It's taking a bunch of steps that weren't ultimately helping, but it didn't occur to me that the solution was to go simpler rather than more complicated. I'll have to keep watch for problem-solving dead-ends like this.

3.67 Exercise 2.58: Processing expressions with infix notation

3.67.1 Question

Suppose we want to modify the differentiation program so that it works with ordinary mathematical notation, in which + and * are infix rather than prefix operators. Since the differentiation program is defined in terms of abstract data, we can modify it to work with different representations of expressions solely by changing the predicates, selectors, and constructors that define the representation of the algebraic expressions on which the differentiator is to operate.

3.67.2 Part 1

Show how to do this in order to differentiate algebraic expressions presented in infix form, such as (x + (3 * (x + (y + 2)))). To simplify the task, assume that + and * always take two arguments and that expressions are fully parenthesized.

3.67.3 Answer 1

```
1    (define (variable? x) (symbol? x))
2    (define (same-variable? v1 v2)
3      (and (variable? v1) (variable? v2) (eq? v1 v2)))
4    (define (sum? x) (and (pair? x) (eq? (cadr x) '+)))
5    (define (addend s) (car s))
6    (define (augend s) (caddr s))
7    (define (product? x) (and (pair? x) (eq? (cadr x) '*)))
8    (define (multiplier p) (car p))
9    (define (multiplicand p) (caddr p))
10   (define (make-sum a1 a2)
11     (cond ((=number? a1 0) a2)
12           ((=number? a2 0) a1)
13           ((and (number? a1) (number? a2))
14            (+ a1 a2))
```

```
15          (else (list a1 '+ a2))))
16    (define (=number? exp num)
17      (and (number? exp) (= exp num)))
18    (define (make-product m1 m2)
19      (cond ((or (=number? m1 0) (=number? m2 0)) 0)
20            ((=number? m1 1) m2)
21            ((=number? m2 1) m1)
22            ((and (number? m1) (number? m2)) (* m1 m2))
23            (else (list m1 '* m2))))
24    (define (deriv exp var)
25      (cond ((number? exp) 0)
26            ((variable? exp) (if (same-variable? exp var) 1 0))
27            ((sum? exp) (make-sum (deriv (addend exp) var)
28                                  (deriv (augend exp) var)))
29            ((product? exp)
30             (make-sum
31               (make-product (multiplier exp)
32                             (deriv (multiplicand exp) var))
33               (make-product (deriv (multiplier exp) var)
34                             (multiplicand exp))))
35            (else
36             (error "unknown expression type: DERIV" exp))))
```

```
1    <<deriv-infix>>
2    (load "mattcheck2.scm")
3    (mattcheck "deriv: basic addition"
4               (deriv '(x + 3) 'x)
5               1)
6    (mattcheck "deriv: basic multiplication"
7               (deriv '(x * y) 'x)
8               'y)
9    (mattcheck "deriv: bad simplification"
10              (deriv '((x * y) * (x + 3)) 'x)
11              '((x * y) + (y * (x + 3)))))
```

```
SUCCEED at deriv: basic addition
SUCCEED at deriv: basic multiplication
SUCCEED at deriv: bad simplification
```

3.67.4   Part 2

The problem becomes substantially harder if we allow standard algebraic nota-
tion, such as (x + 3 * (x + y + 2)), which drops unnecessary parentheses and
assumes that multiplication is done before addition. Can you design appropriate
predicates, selectors, and constructors for this notation such that our derivative
program still works?

3.67.5 Answer 2

Ok, I think I can do the long-form list objective, since that would be a combination of two earlier exercises. But as I write I feel clueless how to make multiplication happen before addition – my first impulse is that this would mean I would need to either:

1. Change how Lisp's evaluation works (obviously overkill)

2. Make statements be evaluated twice, once for multiplication and then once for addition. And I think that would require modifications to `deriv`.

I'll just start working on the long-form feature and see if I can think of anything.

```
(define (variable? x) (symbol? x))
(define (same-variable? v1 v2)
  (and (variable? v1) (variable? v2) (eq? v1 v2)))
(define (sum? x) (and (pair? x) (eq? (cadr x) '+)))
(define (addend s) (car s))
(define (augend s)
  (let ((rest (cddr s)))
    (if (null? (cdr rest))
        (car rest)
        rest)))
(define (product? x) (and (pair? x) (eq? (cadr x) '*)))
(define (multiplier p) (car p))
(define (multiplicand p)
  (let ((rest (cddr p)))
    (if (null? (cdr rest))
        (car rest)
        rest)))
(define (make-sum a1 a2)
  (cond ((=number? a1 0) a2)
        ((=number? a2 0) a1)
        ((and (number? a1) (number? a2))
         (+ a1 a2))
        ((product? a2)
         (list a1 '+ ;; cross your fingers!
               (make-product (multiplier a2)
                             (multiplicand a2))))
        (else (list a1 '+ a2))))
(define (=number? exp num)
  (and (number? exp) (= exp num)))
(define (make-product m1 m2)
  (cond ((or (=number? m1 0) (=number? m2 0)) 0)
        ((=number? m1 1) m2)
        ((=number? m2 1) m1)
        ((and (number? m1) (number? m2)) (* m1 m2))
        (else (list m1 '* m2))))
```

```
36  (define (deriv exp var)
37    (cond ((number? exp) 0)
38          ((variable? exp) (if (same-variable? exp var) 1 0))
39          ((sum? exp) (make-sum (deriv (addend exp) var)
40                                (deriv (augend exp) var)))
41          ((product? exp)
42           (make-sum
43            (make-product (multiplier exp)
44                          (deriv (multiplicand exp) var))
45            (make-product (deriv (multiplier exp) var)
46                          (multiplicand exp))))
47          (else
48           (error "unknown expression type: DERIV" exp))))
```

```
1   <<deriv-algebraic>>
2   (load "mattcheck2.scm")
3   (mattcheck "deriv: basic addition"
4              (deriv '(x + 3) 'x)
5              1)
6   (mattcheck "deriv: basic multiplication"
7              (deriv '(x * y) 'x)
8              'y)
9   (mattcheck "deriv: bad simplification"
10             (deriv '((x * y) * (x + 3)) 'x)
11             '((x * y) + (y * (x + 3))))
12  (mattcheck "deriv: longer addition"
13             (deriv '(x + 3 + 6 + z) 'x)
14             1)
15  (mattcheck "deriv: longer multiplication"
16             (deriv '(x * y * (x + 3)) 'x)
17             '((x * y) + (y * (x + 3))))
18
19  ;; Test cases from:
20  ;; http://community.schemewiki.org/?sicp-ex-2.58
21  (mattcheck "deriv: mult prioritization 1"
22             (deriv '(x + 3 * (x + y + 2)) 'x)
23             4)
24  (mattcheck "deriv: mult prioritization 2"
25             (deriv '(x * (y * (x + 3))) 'x)
26             '((x * y) + (y * (x + 3))))
27  (mattcheck "deriv: mult prioritization 3"
28             (deriv '((x * y) * (x + 3)) 'x)
29             '((x * y) + (y * (x + 3))))
30  ;; Checks from "future Matt"
31  (mattcheck "deriv: mult prioritization 4a"
32             (deriv '(x + 3 + x + y * 2) 'x)
33             2)
34  (mattcheck "deriv: mult prioritization 4b"
```

```
35          (deriv '(y * 2 + x + 3 + x) 'x)
36          2)
```

```
<unknown-location>: warning: possibly unbound variable `mattcheck'
SUCCEED at deriv: basic addition
SUCCEED at deriv: basic multiplication
SUCCEED at deriv: bad simplification
SUCCEED at deriv: longer addition
SUCCEED at deriv: longer multiplication
SUCCEED at deriv: mult prioritization 1
SUCCEED at deriv: mult prioritization 2
SUCCEED at deriv: mult prioritization 3
SUCCEED at deriv: mult prioritization 4a
FAIL at deriv: mult prioritization 4b
expected: 2
returned: (y * 2)
```

Above is my first solution. I modified `make-sum` to check the operator after the current one, see if it was a multiplication, and if so to evaluate it with `make-product` before doing the current addition. The "immutable variable" ways of problem solving makes this pretty easy:

```
1   ;; I want to do f to the first item of x, unless some-condition? in the
    ↪  second
2   ;; item, in which case do g to the 2nd item before x to the first
3   (define (foo x)
4     (if (some-condition? (cdr x))
5         (f (cons (car x)
6                  (g (cdr x))))
7         (f x)))
```

If I were doing this with mutable variables, where I needed to perform these modifications in-place in one data structure, I might not have the tools to solve that problem.

However, I came back to this problem after continuing on ahead so far as Exercise 2.91: division of polynomials, when someone on the Discord asked for help. Upon re-examination I realized that this first solution does *not* solve the problem in an algebraically equivalent way, as you can see by the last two tests I added above. I had assumed that because addition and multiplication were commutative, that I only needed to account for immediately adjacent operations. This was not the case. I must solve the higher-priority operations in a list first before starting any of the lower ones.

TODO:

```
1   (define func-list '(product? sum?))
2   (define priority-list '(* +))
3   (define (solve-in-order exp func-list)
```

```scheme
    (if (< (length exp) 3)
        '()
        (cons (list-index (lambda(i)(eqv? i (cadr exp)))
                          priority-list)
              (find-solution-order (cddr exp)))))
(define (variable? x) (symbol? x))
(define (same-variable? v1 v2)
  (and (variable? v1) (variable? v2) (eq? v1 v2)))
(define (sum? x) (and (pair? x) (eq? (cadr x) '+)))
(define (addend s) (car s))
(define (augend s)
  (let ((rest (cddr s)))
    (if (null? (cdr rest))
        (car rest)
        rest)))
(define (product? x) (and (pair? x) (eq? (cadr x) '*)))
(define (multiplier p) (car p))
(define (multiplicand p)
  (let ((rest (cddr p)))
    (if (null? (cdr rest))
        (car rest)
        rest)))
(define (make-sum a1 a2)
  (cond ((=number? a1 0) a2)
        ((=number? a2 0) a1)
        ((and (number? a1) (number? a2))
         (+ a1 a2))
        ((product? a2)
         (list a1 '+ ;; cross your fingers!
               (make-product (multiplier a2)
                             (multiplicand a2))))
        (else (list a1 '+ a2))))
(define (=number? exp num)
  (and (number? exp) (= exp num)))
(define (make-product m1 m2)
  (cond ((or (=number? m1 0) (=number? m2 0)) 0)
        ((=number? m1 1) m2)
        ((=number? m2 1) m1)
        ((and (number? m1) (number? m2)) (* m1 m2))
        (else (list m1 '* m2))))
(define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp) (if (same-variable? exp var) 1 0))
        ((sum? exp) (make-sum (deriv (addend exp) var)
                              (deriv (augend exp) var)))
        ((product? exp)
         (make-sum
          (make-product (multiplier exp)
```

```
52                          (deriv (multiplicand exp) var))
53              (make-product (deriv (multiplier exp) var)
54                            (multiplicand exp))))
55          (else
56           (error "unknown expression type: DERIV" exp))))
```

```
1   <<deriv-algebraic-fixed>>
2   (load "mattcheck2.scm")
3   (mattcheck "deriv: basic addition"
4             (deriv '(x + 3) 'x)
5             1)
6   (mattcheck "deriv: basic multiplication"
7             (deriv '(x * y) 'x)
8             'y)
9   (mattcheck "deriv: bad simplification"
10            (deriv '((x * y) * (x + 3)) 'x)
11            '((x * y) + (y * (x + 3))))
12  (mattcheck "deriv: longer addition"
13            (deriv '(x + 3 + 6 + z) 'x)
14            1)
15  (mattcheck "deriv: longer multiplication"
16            (deriv '(x * y * (x + 3)) 'x)
17            '((x * y) + (y * (x + 3))))
18
19  ;; Test cases from:
20  ;; http://community.schemewiki.org/?sicp-ex-2.58
21  (mattcheck "deriv: mult prioritization 1"
22            (deriv '(x + 3 * (x + y + 2)) 'x)
23            4)
24  (mattcheck "deriv: mult prioritization 2"
25            (deriv '(x * (y * (x + 3))) 'x)
26            '((x * y) + (y * (x + 3))))
27  (mattcheck "deriv: mult prioritization 3"
28            (deriv '((x * y) * (x + 3)) 'x)
29            '((x * y) + (y * (x + 3))))
30  ;; Checks from "future Matt"
31  (mattcheck "deriv: mult prioritization 4a"
32            (deriv '(x + 3 + x + y * 2) 'x)
33            2)
34  (mattcheck "deriv: mult prioritization 4b"
35            (deriv '(y * 2 + x + 3 + x) 'x)
36            2)
```

3.68   Exercise 2.59: Representing sets

3.68.1   Text definitions

208

```scheme
(define (element-of-set?-manual x set)
  (cond ((null? set) #f)
        ((equal? x (car set)) #t)
        (else (element-of-set? x (cdr set)))))
(define element-of-set? member) ; builtins are faster :)

(define (adjoin-set x set)
  (if (element-of-set? x set)
      set
      (cons x set)))

(define (intersection-set set1 set2)
  (cond ((or (null? set1) (null? set2))
         '())
        ((element-of-set? (car set1) set2)
         (cons (car set1)
               (intersection-set (cdr set1)
                                 set2)))
        (else (intersection-set (cdr set1)
                                set2))))
```

3.68.2   Question

Implement the `union-set` operation for the unordered-list representation of sets.

3.68.3   Answer

`union-set` at a first glance looks like the opposite of `intersection-set`, since the logic looks like "merge two sets, and if an element exists in both than don't include it". However it's actually just avoiding putting a symbol in twice.

```scheme
(define (union-set-rec set1 set2)
  (cond ((or (null? set1) (null? set2))
         set2)
        ((element-of-set? (car set1) set2)
         (union-set-rec (cdr set1)
                        set2))
        (else (cons (car set1)
                    (union-set-rec (cdr set1)
                                   set2)))))
(define (union-set set1 set2)
  (define (iter s result)
    (cond ((null? s)
           (reverse result))
          ((element-of-set? (car s) set2)
           (iter (cdr s) result))
          (else
           (iter (cdr s)
```

```
18                    (cons (car s)
19                          result)))))
20       (append (iter set1 '())
21               set2))
```

```
1    <<sets-txt>>
2    <<union-set>>
3    (load "mattcheck2.scm")
4    (let ((set1 (list 1 2 3 4 5 6))
5          (set2 (list 4 5 6 7 8 9))
6          (set-union (list 1 2 3 4 5 6 7 8 9)))
7      (mattcheck "union-set-rec"
8                 (union-set-rec set1 set2)
9                 set-union)
10     (mattcheck "union-set"
11                (union-set set1 set2)
12                set-union))
```

```
SUCCEED at union-set-rec
SUCCEED at union-set
```

```
1    <<enumerate-interval>>
2    <<sets-txt>>
3    <<union-set>>
4    (load "../mattbench.scm")
5
6    (let ((set1 (enumerate-interval 1 1000))
7          (set2 (enumerate-interval 500 1500)))
8      (define (test)
9        (format #t "~&recursive union-sets: ~a"
10               (cadr (mattbench2 (lambda()(union-set-rec set1 set2))
                       ↪  10000)))
11       (format #t "~&iterative union-sets: ~a"
12               (cadr (mattbench2 (lambda()(union-set set1 set2)) 10000))))
13
14     (test))
```

```
recursive union-sets: 4704343.6974
iterative union-sets: 4755592.2304
```

Man, my "optimized" versions never work. I would blame it on the Scheme compiler but I'm a new programmer so it's probably a skill issue.

3.69 Exercise 2.59: Sets with duplicates

3.69.1 Question

We specified that a set would be represented as a list with no duplicates. Now suppose we allow duplicates. For instance, the set $\{1, 2, 3\}$ could be represented

as the list (`2 3 2 1 3 2 2`). Design procedures `element-of-set?`, `adjoin-se⌋` `t`, `union-set`, and `intersection-set` that operate on this representation. How does the efficiency of each compare with the corresponding procedure for the non-duplicate representation? Are there applications for which you would use this representation in preference to the non-duplicate one?

3.69.2   Answer

`element-of-set` can be left unchanged. The others are a matter of getting sloppy: `adjoin-set` can just be a `cons`, and `union-set` can be an append. Now the remaining question is, how to make `intersection-set` keep the duplicates?

```
1   (define (adjoin-set-dupes x set)
2     (cons x set))
3
4   (define (union-set-dupes set1 set2)
5     (append set1 set2))
6
7   (define (intersection-set-dupes set1 set2)
8     (let ((inter (intersection-set set1 set2)) ;; yes, we're calling the
      ↪   non-duplicate version
9           (union (union-set-dupes set1 set2)))
10      (filter (lambda(x) (element-of-set? x inter))
11              union)))
12  (define (intersection-set-dupes2 set1 set2)
13    (let ((inter (intersection-set set1 set2)) ;; yes, we're calling the
      ↪   non-duplicate version
14          (inter2 (intersection-set set2 set1)))
15      (append inter inter2)))
```

```
1   <<sets-txt>>
2   <<union-set>>
3   <<set-dupes>>
4   (load "mattcheck2.scm")
5   (let ((set1 (list 1 2 3 4 5 6))
6         (set2 (list 4 5 6 7 8 9))
7         (set-union (list 1 2 3 4 5 6 7 8 9))
8         (set-union-dupes
9          (list 1 2 3 4 5 6 4 5 6 7 8 9))
10         (set-intersection (list 4 5 6))
11         (set-intersection-dupes
12          (list 4 5 6 4 5 6)))
13     (mattcheck "union-set-dupes"
14               (union-set-dupes set1 set2)
15               set-union-dupes)
16     (mattcheck "union-set"
17               (union-set set1 set2)
```

211

```
18              set-union)
19      (mattcheck "intersection-set"
20              (intersection-set set1 set2)
21              set-intersection)
22      (mattcheck "intersection-set-dupes"
23              (intersection-set-dupes set1 set2)
24              set-intersection-dupes)
25      (mattcheck "intersection-set-dupes2"
26              (intersection-set-dupes2 set1 set2)
27              set-intersection-dupes))
```

```
 SUCCEED at union-set-dupes
 SUCCEED at union-set
 SUCCEED at intersection-set
 SUCCEED at intersection-set-dupes
 SUCCEED at intersection-set-dupes2
```

```
1   <<enumerate-interval>>
2   <<sets-txt>>
3   <<union-set>>
4   <<set-dupes>>
5   (use-modules (ice-9 format))
6   (load "../mattbench.scm")
7
8   (let ((set1 (enumerate-interval 1 1000))
9         (set2 (enumerate-interval 500 1500)))
10    (define (test)
11      (format #t "~&union-sets: ~a"
12              (cadr (mattbench2 (lambda()(union-set set1 set2)) 10000)))
13      (format #t "~&union-sets-dupes: ~a"
14              (cadr (mattbench2 (lambda()(union-set-dupes set1 set2))
                ↪ 10000)))
15      (format #t "~&intersection-set: ~a"
16              (cadr (mattbench2 (lambda()(intersection-set set1 set2))
                ↪ 10000)))
17      (format #t "~&intersection-set-dupes: ~a"
18              (cadr (mattbench2 (lambda()(intersection-set-dupes set1
                ↪ set2)) 10000)))
19      (format #t "~&intersection-set-dupes2: ~a"
20              (cadr (mattbench2 (lambda()(intersection-set-dupes2 set1
                ↪ set2)) 10000)))
21      )
22
23    (test))
```

```
 union-sets: 4734892.3798
 union-sets-dupes: 40132.52
 intersection-set: 4673425.9196
```

```
intersection-set-dupes: 10325053.8432
intersection-set-dupes2: 10872996.2555
```

So for `union-sets` a significant speedup, while in algorithms that need to check for duplicates like `intersection-set` it's much more time. Also, in a no-duplicate implementation `element-of-set`, it wouldn't be wasting time checking duplicates.

3.70 Exercise 2.61: Ordered sets

3.70.1 Question

Give an implementation of `adjoin-set` using the ordered representation. By analogy with `element-of-set?` show how to take advantage of the ordering to produce a procedure that requires on the average about half as many steps as with the unordered representation.

3.70.2 Answer

```scheme
(define (element-of-set?-ordered x set)
  (cond ((null? set) #f)
        ((= x (car set)) #t)
        ((< x (car set)) #f)
        (else (element-of-set?-ordered x (cdr set)))))

(define (adjoin-set-ordered x set)
  (define (iter checked rest)
    (cond ((null? rest)
             (append checked (list x)))
          ((= x (car rest)) set)
          ((> x (car rest))
           (iter (cons (car rest)
                       checked)
                 (cdr rest)))
          (else (append (reverse checked)
                        (cons x rest)))))
  (iter '() set))
```

```scheme
<<sets-txt>>
<<union-set>>
<<set-ordered>>
(load "mattcheck2.scm")
(let ((set (list 1 2 4 5 6 7 8 9))
      (answer (list 1 2 3 4 5 6 7 8 9)))
  (mattcheck "adjoin-set-ordered"
             (adjoin-set-ordered 3 set)
             answer))
```

```
SUCCEED at adjoin-set-ordered
```

213

3.71 Exercise 2.62: `union-set` ordered

3.71.1 Question

Give a $\Theta(n)$ implementation of `union-set` for sets represented as ordered lists.

3.71.2 Answer

```scheme
(define (union-set-ordered set1 set2)
  (define (iter s1 s2 result)
    (cond ((null? s1)
           (append (reverse result) s2))
          ((null? s2)
           (append (reverse result) s1))
          (else
           (let ((s1a (car s1))
                 (s2a (car s2)))
             (cond ((= s1a s2a)
                    (iter (cdr s1) (cdr s2)
                          (cons s1a result)))
                   ((< s1a s2a)
                    (iter (cdr s1) s2
                          (cons s1a result)))
                   ((> s1a s2a)
                    (iter s1 (cdr s2)
                          (cons s2a result)))))))))
  (iter set1 set2 '()))
```

```scheme
<<sets-txt>>
<<union-set>>
<<set-ordered>>
<<union-set-ordered>>
(load "mattcheck2.scm")
(let ((set1 (list 1 2 3 4 5))
      (set2 (list 4 5 6 7 8 9))
      (answer (list 1 2 3 4 5 6 7 8 9)))
  (mattcheck "union-set-ordered"
             (union-set-ordered set1 set2)
             answer))
```

```
SUCCEED at union-set-ordered
```

```scheme
<<enumerate-interval>>
<<sets-txt>>
<<union-set>>
<<union-set-ordered>>
(use-modules (ice-9 format))
(load "../mattbench.scm")
```

```scheme
;; http://community.schemewiki.org/?sicp-ex-2.62
(define (union-set-alt set1 set2)
  (cond  ((null? set1) set2)
         ((null? set2) set1)
         (else
          (let ((x1 (car set1))
                (x2 (car set2)))
            (cond ((= x1 x2) (cons x1 (union-set-alt (cdr set1) (cdr
              ↪  set2))))
                  ((< x1 x2) (cons x1 (union-set-alt (cdr set1) set2)))
                  (else (cons x2 (union-set-alt set1 (cdr set2)))))))))
(define (union-set-alt2 set1 set2)
  (cond ((null? set1) set2)
        ((null? set2) set1)
        (else
         (let ((x1 (car set1))
               (x2 (car set2)))
           (cons (min x1 x2)
                 (union-set-alt2 (if (> x1 x2)
                                     set1
                                     (cdr set1))
                                 (if (> x2 x1)
                                     set2
                                     (cdr set2))))))))

(let ((set1 (enumerate-interval 1 1000))
      (set2 (enumerate-interval 500 1500)))
  (define (test)
    (format #t "~&union-set: ~a"
            (cadr (mattbench2 (lambda()(union-set set1 set2)) 10000)))
    (format #t "~&union-set-ordered: ~a"
            (cadr (mattbench2 (lambda()(union-set-ordered set1 set2))
              ↪  50000)))
    (format #t "~&union-set-alt: ~a"
            (cadr (mattbench2 (lambda()(union-set-alt set1 set2))
              ↪  500000)))
    (format #t "~&union-set-alt2: ~a"
            (cadr (mattbench2 (lambda()(union-set-alt2 set1 set2))
              ↪  500000)))
    )

  (test))
```

```
union-set: 4758999.4984
union-set-ordered: 107784.3673
union-set-alt: 45441.7717
union-set-alt2: 66262.4295
```

3.72 Exercise 2.63: binary trees

3.72.1 Text definitions

```
1  (define (entry tree) (car tree))
2  (define (left-branch tree) (cadr tree))
3  (define (right-branch tree) (caddr tree))
4  (define (make-tree entry left right)
5    (list entry left right))
```

3.72.2 Question A

Each of the following two procedures converts a binary tree to a list.

```
1  <<make-tree>>
2  (define (tree->list-1 tree)
3    (if (null? tree)
4        '()
5        (append (tree->list-1 (left-branch tree))
6                (cons (entry tree)
7                      (tree->list-1
8                       (right-branch tree))))))
9  (define (tree->list-2 tree)
10   (define (copy-to-list tree result-list)
11     (if (null? tree)
12         result-list
13         (copy-to-list (left-branch tree)
14                       (cons (entry tree)
15                             (copy-to-list
16                              (right-branch tree)
17                              result-list)))))
18   (copy-to-list tree '()))
```

1. Do the two procedures produce the same result for every tree? If not, how do the results differ? What lists do the two procedures produce for the trees in Figure 2.16?

3.72.3 Answer A

First let's check whether the arrangement of the input tree impacts the output list. I'd like a couple functions for generating lists.

```
1  (define (rightward-tree list)
2    (if (null? list)
3        '()
4        (make-tree (car list) '()
5                   (rightward-tree (cdr list)))))
6  (define (leftward-tree list)
7    (define (rec list)
```

```
 8        (if (null? list)
 9            '()
10            (make-tree (car list) (rec (cdr list))
11                       '()))))
12      (rec list))
13    (define (nested-tree list)
14      (define (rec list)
15        (cond ((= 1 (length list))
16               (make-tree (car list) '() '()))
17              ((= 2 (length list))
18               (make-tree (cadr list)
19                          (make-tree (car list) '() '())
20                          '()))
21              (else
22               (let ((halfway (truncate/ (length list)
23                                         2)))
24                 (make-tree (list-ref list halfway)
25                            (rec (list-head list halfway))
26                            (rec (list-tail list (1+ halfway))))))))
27      (rec list))
```

```
 1    <<echo>>
 2    <<tree-to-list>>
 3    <<tree-makers>>
 4
 5    (let* ((nil '())
 6           (1-to-4 (list 1 2 3 4))
 7           (tree-right
 8            (rightward-tree 1-to-4))
 9           (tree-left
10            (leftward-tree 1-to-4))
11           (tree-middle
12            (nested-tree (iota 10))))
13      (echo "tree-right:" tree-right)
14      (echo "tree-right list 1:" (tree->list-1 tree-right))
15      (echo "tree-right list 2:" (tree->list-2 tree-right))
16      (newline)
17      (echo "tree-left:" tree-left)
18      (echo "tree-left list 1:" (tree->list-1 tree-left))
19      (echo "tree-left list 2:" (tree->list-2 tree-left))
20      (newline)
21      (echo "tree-middle:" tree-middle)
22      (echo "tree-middle list 1:" (tree->list-1 tree-middle))
23      (echo "tree-middle list 2:" (tree->list-2 tree-middle)))
```

```
tree-right: (1 () (2 () (3 () (4 () ()))))
tree-right list 1: (1 2 3 4)
tree-right list 2: (1 2 3 4)
```

```
tree-left: (1 (2 (3 (4 () ()) ()) ()) ())
tree-left list 1: (4 3 2 1)
tree-left list 2: (4 3 2 1)

tree-middle: (5 (2 (1 (0 () ()) ()) ()) (4 (3 () ()) ()))) (8 (7 (6 () ()) ()) (9 () ()))))
tree-middle list 1: (0 1 2 3 4 5 6 7 8 9)
tree-middle list 2: (0 1 2 3 4 5 6 7 8 9)
```

So, the lists are the same for both.

Fun fact: Emacs org-babel has suddenly stopped accepting the λ character in source code.

3.72.4   Question B

Do the two procedures have the same order of growth in the number of steps required to convert a balanced tree with n elements to a list? If not, which one grows more slowly?

3.72.5   Answer B

```
1   ;; tree->list-1 evaluating the following tree:
2   ;;       7
3   ;;      / \
4   ;;    3   9
5   ;;   / \   \
6   ;;  1   5   11
7   (append (append (append nil
8                           (cons 1
9                                 nil))
10                  (cons 3
11                        (append nil
12                                (cons 5
13                                      nil))))
14          (cons 7
15                (append nil
16                        (cons 9
17                              (append nil
18                                      (cons 11
19                                            nil))))))
20  ;; holy cow that is wasteful. So many appends of nothing.
21
22  ;; tree->list-2 evaluation (iterative)
23  (copy-to-list [full tree] nil)
24  (copy-to-list [tree 3 1 5]
25                (cons 7
26                      (copy-to-list
27                      [tree 9 11]
```

```
28                            nil)))
29   (copy-to-list [tree 3 1 5]
30                 (cons 7
31                       (copy-to-list
32                        nil
33                        (cons 9
34                              (copy-to-list
35                               [tree 11]
36                               nil)))))
37   (copy-to-list [tree 3 1 5]
38                 (cons 7
39                       (copy-to-list
40                        nil
41                        (cons 9
42                              (copy-to-list
43                               nil
44                               (cons 11
45                                     (copy-to-list nil nil)))))))
46   (copy-to-list [tree 3 1 5]
47                 '(7 9 11))
48   (copy-to-list [tree 1]
49                 (cons 3
50                       (copy-to-list
51                        nil
52                        (cons 5
53                              (copy-to-list
54                               nil
55                               '(7 9 11))))))
56   (copy-to-list
57    nil
58    (cons 1
59          (copy-to-list
60           nil
61           '(3 5 7 9 11))))
62   '(1 3 5 7 9 11)
```

So, assuming the evaluation isn't as slow as operations like `append`, the second is definitely faster – it's basically just evaluating to a series of `cons` statements. The internet suggests that the first is $\Theta(n \log n)$ while the second is $\Theta(n)$.

3.73   Exercise 2.64: Making a balanced binary tree

The following procedure `list->tree` converts an ordered list to a balanced binary tree. The helper procedure `partial-tree` takes as arguments an integer n and list of at least n elements and constructs a balanced tree containing the first n elements of the list. The result returned by `partial-tree` is a pair (formed with `cons`) whose `car` is the constructed tree and whose `cdr` is the list of elements not included in the tree.

219

```
1   <<make-tree>>
2   (define (list->tree elements)
3     (car (partial-tree elements (length elements))))
4   (define (partial-tree elts n)
5     (if (= n 0)
6         (cons '() elts)
7         (let ((left-size (quotient (- n 1) 2)))
8           (let ((left-result
9                   (partial-tree elts left-size)))
10            (let ((left-tree (car left-result))
11                  (non-left-elts (cdr left-result))
12                  (right-size (- n (+ left-size 1))))
13              (let ((this-entry (car non-left-elts))
14                    (right-result
15                     (partial-tree
16                      (cdr non-left-elts)
17                      right-size)))
18                (let ((right-tree (car right-result))
19                      (remaining-elts
20                       (cdr right-result)))
21                  (cons (make-tree this-entry
22                                   left-tree
23                                   right-tree)
24                        remaining-elts))))))))
```

3.73.1   Question A

Write a short paragraph explaining as clearly as you can how `partial-tree` works. Draw the tree produced by `list->tree` for the list (**1 3 5 7 9 11**).

3.73.2   Answer A

For my own sake, I'll reorganize this with **let***.

```
1   (define (partial-tree elts n)
2     (if (= n 0)
3         (cons '() elts)
4         (let* ((left-size (quotient (- n 1) 2))
5                (left-result (partial-tree
6                              elts
7                              left-size))
8                (left-tree (car left-result))
9                (non-left-elts (cdr left-result))
10               (right-size (- n (+ left-size 1)))
11               (this-entry (car non-left-elts))
12               (right-result (partial-tree
13                              (cdr non-left-elts)
14                              right-size))
```

```
15              (right-tree (car right-result))
16              (remaining-elts (cdr right-result)))
17          (cons (make-tree this-entry
18                           left-tree
19                           right-tree)
20              remaining-elts))))
21
```

Say we're evaluating '(**1 3 5 7 9 11**). We define `left-size` as $\lfloor n/2 \rfloor$. `pa⌋` `rtial-tree` recurses until hitting $n = 0$, at which point it returns the starting list with '() prepended to it. This is used as the empty left branch for the second-to-the-bottom row of the tree, in this case for the tree node of **1**. That empty tree branch is saved in `left-tree`, `this-entry` (**1**) is defined, then we recurse to the right branch, finding a bottom to place **3**. Finally, the tree with only 3 is returned, it becomes the right branch off **1**, making (**make-tree 1** ⌋ '() (**make-tree 3** '() '())). This gets `cons`'d to the remaining list, '(**5 7 9 11**). Upon returning the new list, this will become the `left-tree` of its parent process. All in all, this is a recursive algorithm that kind of solves itself with minimal logic. Note its evaluation always comes back to `cons` operations.

This is very elegant. However, I am left wondering how clear this is in comparison with some C routine that does the same thing.
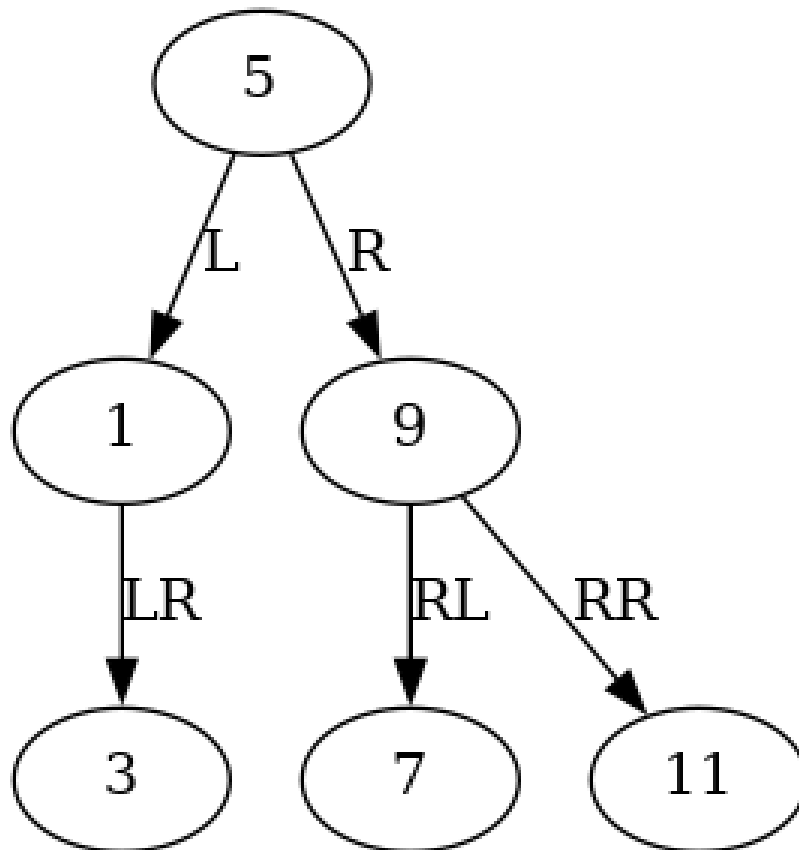
3.73.3 Question B

What is the order of growth in the number of steps required by `list->tree` to convert a list of n elements?

3.73.4 Answer B

I'm going to estimate that it's $\Theta(n)$ because of the `cons` operations being the cornerstone of the procedure. The internet appears to agree in this case.

3.74 Exercise 2.65: Sets as binary trees

3.74.1 Textbook Definitions

```
(define (element-of-set?-tree x set)
  (cond ((null? set) #f)
        ((= x (entry set)) #t)
```

```
4            ((< x (entry set))
5             (element-of-set?-tree
6              x
7              (left-branch set)))
8            ((> x (entry set))
9             (element-of-set?-tree
10              x
11              (right-branch set)))))
12   (define (adjoin-set-tree x set)
13     (cond ((null? set) (make-tree x '() '()))
14           ((= x (entry set)) set)
15           ((< x (entry set))
16            (make-tree
17             (entry set)
18             (adjoin-set-tree x (left-branch set))
19             (right-branch set)))
20           ((> x (entry set))
21            (make-tree
22             (entry set)
23             (left-branch set)
24             (adjoin-set-tree x (right-branch set))))))
```

3.74.2 Question

Use the results of Exercise 2.63 and Exercise 2.64 to give $\Theta(n)$ implementations
of `union-set` and `intersection-set` for sets implemented as (balanced) binary
trees.

3.74.3 Answer

Ok, now that's stretching my brain for sure. Hmm...

I think I need to look closer at how `tree->list` works.

```
1    (define (tree->list-debug tree)
2      (define (copy-to-list tree result-list)
3        (if (null? tree)
4            result-list
5            (let ((result
6                   (begin (echo ">> entering tree at" (entry tree) "results"
                    ↪   result-list)
7                          (copy-to-list (left-branch tree)
8                                        (begin (echo "we'll cons" (entry
                                        ↪   tree))
9                                               (let ((pair
10                                                     (cons (entry tree)
11                                                           (copy-to-list
12                                                            (right-branch
                                                             ↪   tree)
```

223

```
13                                                          result-list))))
14                                              (echo " cons'd" pair)
15                                              pair))))))
16          (echo "<< leaving tree at" (entry tree) "results" result-list)
17          result)))
18    (copy-to-list tree '())))
```

```
>> entering tree at 5 results ()
we'll cons 5
>> entering tree at 9 results ()
we'll cons 9
>> entering tree at 11 results ()
we'll cons 11
  cons'd (11)
<< leaving tree at 11 results ()
  cons'd (9 11)
>> entering tree at 7 results (9 11)
we'll cons 7
  cons'd (7 9 11)
<< leaving tree at 7 results (9 11)
<< leaving tree at 9 results ()
  cons'd (5 7 9 11)
>> entering tree at 1 results (5 7 9 11)
we'll cons 1
>> entering tree at 3 results (5 7 9 11)
we'll cons 3
  cons'd (3 5 7 9 11)
<< leaving tree at 3 results (5 7 9 11)
  cons'd (1 3 5 7 9 11)
<< leaving tree at 1 results (5 7 9 11)
<< leaving tree at 5 results ()
(1 3 5 7 9 11)
```

What's really holding me up is the $\Theta(n)$ complexity. Since I'm probably not going to figure this out easily, let's try doing it the "wrong" way first, by using previously made algorithms.

```
1    <<make-tree>>
2    <<list-to-tree>>
3    <<tree-to-list>>
4    <<tree-sets-txt>>
5
6    (define (tree->list-debug tree)
7      (define (copy-to-list tree result-list)
8        (if (null? tree)
9            result-list
10           (let ((result
```

```scheme
                  (begin (echo ">> entering tree at" (entry tree) "results"
              ↪  result-list)
                        (copy-to-list (left-branch tree)
                                      (begin (echo "we'll cons" (entry
                                      ↪  tree))
                                            (let ((pair
                                                  (cons (entry tree)
                                                        (copy-to-list
                                                         (right-branch
                                                         ↪  tree)
                                                         result-list))))
                                              (echo "  cons'd" pair)
                                              pair))))))
          (echo "<< leaving tree at" (entry tree) "results" result-list)
          result)))
    (copy-to-list tree '())))

  (define (union-set-tree-wrong1 set1 set2)
    (define (copy-to-list tree held-tree result-list)
      (define (swap-trees)
        (copy-to-list held-tree tree result-list))
      (define (advance-tree-no-cons)
        (copy-to-list (left-branch tree)
                      held-tree
                      (copy-to-list
                       (right-branch tree)
                       held-tree
                       result-list)))
      (define (advance-tree-cons)
        (copy-to-list (left-branch tree)
                      held-tree
                      (cons (entry tree)
                            (copy-to-list
                             (right-branch tree)
                             held-tree
                             result-list))))
      (define (advance-both-cons)
        (copy-to-list (left-branch tree)
                      held-tree
                      (cons (entry tree)
                            (copy-to-list
                             (right-branch tree)
                             held-tree
                             result-list))))
      (cond ((and (null? tree)
                  (null? held-tree))
             result-list)
            ((null? tree)
```

```
56              (swap-trees))
57           ((= (entry tree)
58               (car result-list))
59            (advance-tree-no-cons))
60           ((= (entry tree) (entry held-tree))
61            (advance-both-cons))
62           ((< (entry tree) (entry held-tree))
63            (advance-tree-cons))
64           ((> (entry tree) (entry held-tree))
65            (swap-trees))))
66     (copy-to-list set1 set2 '()))

67
68  (define (union-set-tree-wrong2 set1 set2)
69    (define (copy-to-tree tree result-tree)
70      (if (null? tree)
71          result-tree
72          (copy-to-tree (left-branch tree)
73                        (adjoin-set-tree (entry tree)
74                                         (copy-to-tree
75                                          (right-branch tree)
76                                          result-tree)))))
77    (copy-to-tree set1 set2))

78
79  (define (intersection-set-tree-wrong1 set1 set2)
80    (define (copy-to-tree tree result-tree)
81      (if (null? tree)
82          result-tree
83          (let ((adjoin-or-not
84                 (lambda (rest)
85                   (if (element-of-set?-tree (entry tree)
86                                             set2)
87                       (adjoin-set-tree (entry tree)
88                                        rest)
89                       rest))))
90            (copy-to-tree (left-branch tree)
91                          (adjoin-or-not
92                           (copy-to-tree
93                            (right-branch tree)
94                            result-tree))))))
95      (copy-to-tree set1 '()))
```

```
1  <<sets-as-trees>>
2  <<echo>>
3  (load "mattcheck2.scm")
4  (let ((set1 (list->tree '(1 3 5 7 9 11)))
5  ;; (5 (1 () (3 () ())) (9 (7 () ()) (11 () ())))
6        (set2 (list->tree '(2 4 6 7 8 11)))
7  ;; (6 (2 () (4 () ())) (8 (7 () ()) (11 () ())))
```

226

```
8            (union (list->tree '(1 2 3 4 5 6 7 8 9 11)))
9  ;; (5 (2 (1 () ()) (3 () (4 () ()))) (8 (6 () (7 () ())) (9 () (11 ()
   ↪  ())))))
10           (union-unbal '(6 (2 (1 () ()) (4 (3 () ()) (5 () ()))) (8 (7 ()
   ↪  ()) (11 (9 () ()) ()))))
11           (int-unbal '(11 (7 () ()) ()))
12           (int (list->tree '(7 11))))
13 ;; (7 () (11 () ()))
14    (mattcheck "element-of-set?-tree true"
15               (element-of-set?-tree 1 set1)
16               #t)
17    (mattcheck "element-of-set?-tree true"
18               (element-of-set?-tree 7 set1)
19               #t)
20    (mattcheck "element-of-set?-tree false"
21               (element-of-set?-tree 2 set1)
22               #f)
23    (mattcheck "union-set-tree-wrong2"
24               (union-set-tree-wrong2 set1 set2)
25               union-unbal)
26    (mattcheck "intersection-set-tree-wrong1"
27               (intersection-set-tree-wrong1 set1 set2)
28               int-unbal)
29    (echo (tree->list-debug set1)))
```

```
SUCCEED at element-of-set?-tree true
SUCCEED at element-of-set?-tree true
SUCCEED at element-of-set?-tree false
SUCCEED at union-set-tree-wrong2
SUCCEED at intersection-set-tree-wrong1
```

I can't really think of a better way to do it. Time to look up the answer!

And... To my surprise, the answer most internet people have given is what I called the "wrong" one. In their case, using tree->list and list->tree to apply the list-based union-set and intersection-set, saying it's $\Theta(n)$. I also notice that they didn't use element-of-set? and adjoin-to-set like I did.

Let's compare.

```
1  (use-modules (ice-9 format))
2  (load "../mattbench.scm")
3  <<sets-txt>>
4  <<sets-as-trees>>
5  <<enumerate-interval>>
6  (define tree->list tree->list-2)
7  (define (union-set set1 set2)
8    (cond ((or (null? set1) (null? set2))
9           set2)
10          ((element-of-set? (car set1) set2)
```

```
11            (union-set (cdr set1)
12                           set2))
13         (else (cons (car set1)
14                     (union-set (cdr set1)
15                                    set2)))))

16

17  ;; from http://community.schemewiki.org/?sicp-ex-2.65
18  (define (union-set-l2t set1 set2)
19    (cond ((null? set1) set2)
20          ((null? set2) set1)
21          (else (list->tree (union-set (tree->list set1) (tree->list
            ↪  set2))))))

22

23  (define (intersection-set-l2t set1 set2)
24    (cond ((null? set1) '())
25          ((null? set2) '())
26          (else (list->tree (intersection-set (tree->list set1)
            ↪  (tree->list set2))))))

27

28  (define (make-2sets max)
29    (let* ((a-start 0)
30           (a-end 0.6)
31           (b-start 0.4)
32           (b-end 1))
33      (cons (list->tree
34              (enumerate-interval
35               0
36               (inexact->exact (* a-end max))))
37             (list->tree
38              (enumerate-interval
39               (inexact->exact (* b-start max))
40               (inexact->exact (* b-end max)))))))

41

42  (define (test-with 2sets repeats)
43    (format #t "~&union-set-tree-wrong2: ~a"
44            (cadr (mattbench2
45                   (lambda()(union-set-tree-wrong2
46                            (car 2sets)
47                            (cdr 2sets)))
48                   repeats)))
49    (format #t "~&union-set-l2t: ~a"
50            (cadr (mattbench2
51                   (lambda()(union-set-l2t (car 2sets)
52                                          (cdr 2sets)))
53                   repeats)))
54    (format #t "~&intersection-set-tree-wrong1: ~a"
55            (cadr (mattbench2
56                   (lambda()(intersection-set-tree-wrong1
```

```
57                      (car 2sets)
58                      (cdr 2sets)))
59                  repeats)))
60    (format #t "~&intersection-set-l2t: ~a"
61            (cadr (mattbench2
62                    (lambda()(intersection-set-l2t (car 2sets)
63                                                   (cdr 2sets)))
64                  repeats))))
65  (format #t "~&~a" (make-2sets 10))
66  (define base-repeats 10000)
67  (test-with (make-2sets 100)
68            base-repeats)
69  (test-with (make-2sets 1000)
70            (/ base-repeats 2))
71  (test-with (make-2sets 10000)
72            (/ base-repeats 4))
```

```
union-set-tree-wrong2: 159831.3273
union-set-l2t: 49633.5962
intersection-set-tree-wrong1: 42577.4905
intersection-set-l2t: 33261.5268
union-set-tree-wrong2: 12648028.8732
union-set-l2t: 2196015.1536
intersection-set-tree-wrong1: 3165935.4476
intersection-set-l2t: 2015603.052
union-set-tree-wrong2: 1393227860.116
union-set-l2t: 195546567.7616
intersection-set-tree-wrong1: 361923468.7844
intersection-set-l2t: 195366120.9524
```

time taken to merge sets
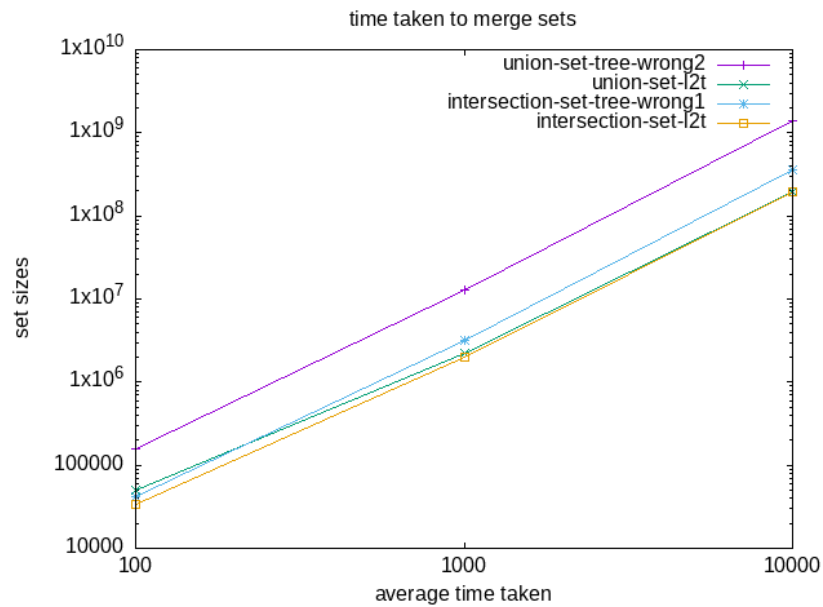
Not really enough datapoints, but you can at least see that the `l2t` variants are faster.

3.75 Exercise 2.66: binary tree `lookup`

3.75.1 Question

Implement the `lookup` procedure for the case where the set of records is structured as a binary tree, ordered by the numerical values of the keys.

3.75.2 Answer

First let's define what these records look like.

```
(define (make-record key value)
  (cons key value))
(define (key record)
  (car record))
(define (value record)
  (cdr record))
```

```
<<make-tree>>
<<make-record>>
(define (lookup given-key tree-of-records)
  (cond ((null? tree-of-records) #f)
        ((= given-key
```

```
6              (key (entry tree-of-records)))
7            (entry tree-of-records))
8          ((< given-key
9              (key (entry tree-of-records)))
10          (lookup
11           given-key
12           (left-branch tree-of-records)))
13         ((> given-key
14              (key (entry tree-of-records)))
15          (lookup
16           given-key
17           (right-branch tree-of-records)))))))
```

```
1   <<lookup-tree>>
2   <<list-to-tree>>
3   <<echo>>
4   (load "mattcheck2.scm")
5   (let ((set1 (list->tree
6                (list (make-record 1 'a)
7                      (make-record 3 'b)
8                      (make-record 5 'c)
9                      (make-record 7 'd)
10                     (make-record 9 'e)
11                     (make-record 11 'f)))))
12     (mattcheck "lookup retrieves correct record"
13                (lookup 9 set1)
14                (make-record 9 'e)))
```

```
SUCCEED at lookup retrieves correct record
```

3.76 Exercise 2.67: decoding Huffman tree messages

3.76.1 Text definitions

```
1   ;; huffman-trees-txt
2   (define (make-leaf symbol weight)
3     (list 'leaf
4           symbol
5           weight))
6   (define (leaf? object)
7     (eq? (car object)
8          'leaf))
9   (define (symbol-leaf x)
10    (cadr x))
11  (define (weight-leaf x)
12    (caddr x))
13
14  (define (make-code-tree left right)
```

```
15      (list left
16            right
17            (append (symbols left) (symbols right))
18            (+ (weight left) (weight right))))
19
20  (define (left-branch  tree)
21    (car tree))
22  (define (right-branch tree)
23    (cadr tree))
24  (define (symbols tree)
25    (if (leaf? tree)
26        (list (symbol-leaf tree))
27        (caddr tree)))
28  (define (weight tree)
29    (if (leaf? tree)
30        (weight-leaf tree)
31        (cadddr tree)))
32  (define (decode bits tree)
33    (define (decode-1 bits current-branch)
34      (if (null? bits)
35          '()
36          (let ((next-branch
37                 (choose-branch (car bits) current-branch)))
38            (if (leaf? next-branch)
39                (cons (symbol-leaf next-branch)
40                      (decode-1 (cdr bits) tree))
41                (decode-1 (cdr bits) next-branch)))))
42    (decode-1 bits tree))
43  (define (choose-branch bit branch)
44    (cond ((= bit 0) (left-branch branch))
45          ((= bit 1) (right-branch branch))
46          (else (error "bad bit: CHOOSE-BRANCH" bit))))
47  (define (adjoin-set x set)
48    (cond ((null? set) (list x))
49          ((< (weight x) (weight (car set))) (cons x set))
50          (else (cons (car set)
51                      (adjoin-set x (cdr set))))))
52  (define (make-leaf-set pairs)
53    (if (null? pairs)
54        '()
55        (let ((pair (car pairs)))
56          (adjoin-set (make-leaf (car pair)
57                                 (cadr pair))
58                      (make-leaf-set (cdr pairs))))))
```

3.76.2 Question

Define an encoding tree and a sample message. Use the `decode` procedure to decode the message, and give the result.

3.76.3 Answer

```
1   <<huffman-trees-txt>>
2   <<echo>>
3   (define sample-tree
4     (make-code-tree (make-leaf 'A 4)
5                     (make-code-tree
6                      (make-leaf 'B 2)
7                      (make-code-tree
8                       (make-leaf 'D 1)
9                       (make-leaf 'C 1)))))
10  (define whos-on-first-tree
11    (make-code-tree
12     (make-leaf 'who 8)
13                    (make-code-tree
14                     (make-leaf 'what 4)
15                     (make-code-tree
16                      (make-leaf 'Idontknow 2)
17                      (make-leaf 'why 2)))))
18  (define sample-message '(0 1 1 0 0 1 0 1 0 1 1 1 0))
19  (define whos-on-first-message '(0 1 0 1 1 0 1 1 1))
20  (echo (decode sample-message sample-tree))
21  (echo (decode whos-on-first-message whos-on-first-tree))
```

```
(A D A B B C A)
(who what Idontknow why)
```

3.77 Exercise 2.68: encoding Huffman tree messages

3.77.1 Question

The `encode` procedure takes as arguments a message and a tree and produces the list of bits that gives the encoded message.

```
1   ;; encode-txt
2   (define (encode message tree)
3     (if (null? message)
4         '()
5         (append (encode-symbol (car message) tree)
6                 (encode (cdr message) tree))))
```

`encode-symbol` is a procedure, which you must write, that returns the list of bits that encodes a given symbol according to a given tree. You should design `encode-symbol` so that it signals an error if the symbol is not in the tree at all. Test your procedure by encoding the result you obtained in Exercise 2.67 with the sample tree and seeing whether it is the same as the original sample message.

3.77.2 Answer

```
1  ;; encode-symbol
2  (define element-of-set? member)
3  (define (encode-symbol symbol tree)
4    (define (rec t)
5      (cond ((and (leaf? t)
6                  (eq? (symbol-leaf t)
7                       symbol))
8             '())
9            ((element-of-set? symbol
10                             (symbols (left-branch t)))
11            (cons 0
12                  (rec (left-branch t))))
13           ((element-of-set? symbol
14                             (symbols (right-branch t)))
15            (cons 1
16                  (rec (right-branch t))))
17           (else (error "encode-symbol: logic error"))))
18    (if (element-of-set? symbol (symbols tree))
19        (rec tree)
20        (error "encode-symbol: symbol not in tree")))
```

```
1  <<huffman-trees-txt>>
2  <<encode-txt>>
3  <<encode-symbol>>
4  (load "mattcheck2.scm")
5  (define sample-tree
6    (make-code-tree (make-leaf 'A 4)
7                    (make-code-tree
8                     (make-leaf 'B 2)
9                     (make-code-tree
10                     (make-leaf 'D 1)
11                     (make-leaf 'C 1)))))
12 (define whos-on-first-tree
13   (make-code-tree
14    (make-leaf 'who 8)
15                   (make-code-tree
16                    (make-leaf 'what 4)
17                    (make-code-tree
18                     (make-leaf 'Idontknow 2)
19                     (make-leaf 'why 2)))))
20 (define sample-message '(0 1 1 0 0 1 0 1 0 1 1 1 0))
21 (define whos-on-first-message '(0 1 0 1 1 0 1 1 1))
22 (mattcheck "sample encoded message"
23            sample-message
24            (encode '(A D A B B C A)
25                    sample-tree))
```

```
26   (mattcheck "who's on first encoded message"
27             whos-on-first-message
28             (encode '(who what Idontknow why)
29                     whos-on-first-tree))
```

```
SUCCEED at sample encoded message
SUCCEED at who's on first encoded message
```

3.78 Exercise 2.69: Generating Huffman trees

3.78.1 Question

The following procedure takes as its argument a list of symbol-frequency pairs (where no symbol appears in more than one pair) and generates a Huffman encoding tree according to the Huffman algorithm.

```
1   (define (generate-huffman-tree pairs)
2     (successive-merge (make-leaf-set pairs)))
```

make-leaf-set is the procedure given above that transforms the list of pairs into an ordered set of leaves. successive-merge is the procedure you must write, using make-code-tree to successively merge the smallest-weight elements of the set until there is only one element left, which is the desired Huffman tree. (This procedure is slightly tricky, but not really complicated. If you find yourself designing a complex procedure, then you are almost certainly doing something wrong. You can take significant advantage of the fact that we are using an ordered set representation.)

3.78.2 Answer

This one took a while. The whole in-place-reorganization of the list seems tricky, since adding branches together makes a new branch which is higher value than the later entries. In a C language I'd be solving the problem by "moving" a play-head forward and back, but here I'll have to coerce the evaluator into doing it.

```
1    ;; generate-huffman-tree
2    (use-srfis '(1))
3    (define element-of-set? member)
4    (define (generate-huffman-tree pairs)
5      (successive-merge (make-leaf-set pairs)))
6
7    (define (successivemerge-iter delayed leaves)
8      (let ((dl (length delayed))
9            (ll (length leaves)))
10       (cond ((= ll 0)
11              (successivemerge-iter '()
12                                    delayed))
```

```
13              ((= dl 0)
14               (if (= ll 1)
15                   (car leaves)
16                   (successivemerge-iter (cons (car leaves) '())
17                                         (cdr leaves))))
18              ((and (= ll 1)
19                    (= dl 1))
20               (make-code-tree (car delayed)
21                               (car leaves)))
22              ((≤ (weight (car delayed))
23                  (weight (car leaves)))
24               (successivemerge-iter (cdr delayed)
25                                     (cons (make-code-tree (car delayed)
26                                                           (car leaves))
27                                           (cdr leaves))))
28              ((> (weight (car delayed))
29                  (weight (car leaves)))
30               (successivemerge-iter (cons (car leaves)
31                                           delayed)
32                                     (cdr leaves))))))

34    (define (successive-merge leaves)
35      (successivemerge-iter (cons (car leaves) '())
36                            (cdr leaves)))
```

```
1    <<huffman-trees-txt>>
2    <<encode-txt>>
3    <<encode-symbol>>
4    <<generate-huffman-tree>>
5    <<echo>>
6    (load "mattcheck2.scm")

8    (let ((pairs
9           (make-leaf-set '((A 8)(B 3)(C 1)(D 1)
10                           (E 1)(F 1)(G 1)(H 1))))
11         (answer
12          (make-code-tree (make-leaf 'A 8)
13                          (make-code-tree
14                           (make-code-tree
15                            (make-leaf 'B 3)
16                            (make-code-tree
17                             (make-leaf 'C 1)
18                             (make-leaf 'D 1)))
19                           (make-code-tree
20                            (make-code-tree
21                             (make-leaf 'E 1)
22                             (make-leaf 'F 1))
23                            (make-code-tree
```

```
24                          (make-leaf 'G 1)
25                          (make-leaf 'H 1)))))))
26     (mattcheck "successive-merge"
27               (successive-merge pairs)
28               answer)
29     (echo (successive-merge pairs)))
```

```
FAIL at successive-merge
expected: ((leaf A 8) (((leaf B 3) ((leaf C 1) (leaf D 1) (C D) 2) (B C D) 5) (# …) …) …)
returned: (((((leaf H 1) (leaf G 1) (H G) 2) ((leaf F 1) (leaf E 1) (F E) 2) (…) …) …) …)
(((((leaf H 1) (leaf G 1) (H G) 2) ((leaf F 1) (leaf E 1) (F E) 2) (H G F E) 4)
  (((leaf D 1) (leaf C 1) (D C) 2) (leaf B 3) (D C B) 5) (H G F E D C B) 9)
    (leaf A 8) (H G F E D C B A) 17)
```

And the tree is *backwards*. But it is correct. Sigh.

Time to be embarrassed by the internet's solution.

```
1    ;; see 2.67
2    (define (adjoin-set x set)
3      (cond ((null? set) (list x))
4            ((< (weight x) (weight (car set)))
5             (cons x set))
6            (else (cons (car set)
7                        (adjoin-set x (cdr set))))))
8    ;; for my own edification
9    (define (adjoin-set-iter x set)
10     (define (iter less more)
11       (cond ((null? more)
12              (reverse (cons x less)))
13             ((< (weight x) (weight (car more)))
14              (append (reverse (cons x less))
15                      more))
16             (else (adjoin-set-iter (cons (car more) less)
17                                    (cdr more)))))
18
19     (iter '() set))
20
21   ;; https://codereview.stackexchange.com/a/117980
22   (define (successive-merge-small leaves)
23     (if (null? (cdr leaves))
24         (car leaves)
25         (successive-merge-small
26          (adjoin-set
27           (make-code-tree (car leaves)
28                           (cadr leaves))
29           (cddr leaves)))))
```

Comparing my above code with the code for `adjoin-set`, mine is definitely
unnecessarily complicated. How does the performance stack up?

237

```
Mine: 4453.0773347
Theirs: 4554.0141235
```

So for all that unnecessary complexity it's still about the same.

The lesson I keep failing to learn is remembering past work (i.e. `adjoin-set`), since this book loves to integrate past exercises.

3.79 Exercise 2.70: encoding 1950s rock songs

3.79.1 Question

The following eight-symbol alphabet with associated relative frequencies was designed to efficiently encode the lyrics of 1950s rock songs. (Note that the symbols of an alphabet need not be individual letters.)

A	2	NA	16
BOOM	1	SHA	3
GET	2	YIP	9
JOB	2	WAH	1

Use `generate-huffman-tree` to generate a corresponding Huffman tree, and use `encode` to encode the following message:

```
Get a job
Sha na na na na na na na na

Get a job
Sha na na na na na na na na

Wah yip yip yip yip
yip yip yip yip yip
Sha boom
```

How many bits are required for the encoding? What is the smallest number of bits that would be needed to encode this song if we used a fixed-length code for the eight-symbol alphabet?

3.79.2 Answer

```
1   (define hippie-set
2     '((NA 16)(YIP 9)(SHA 3)(A 2)
3       (GET 2)(JOB 2)(BOOM 1)(WAH 1)))
4   (define hippie-tree
5     (generate-huffman-tree hippie-set))
6   (define hippie-plaintext
7     '(GET A JOB
8       SHA NA NA NA NA NA NA NA NA
9       GET A JOB
10      SHA NA NA NA NA NA NA NA NA
```

```
11        WAH YIP YIP YIP YIP
12        YIP YIP YIP YIP YIP
13        SHA BOOM))
14    (define hippie-encoded
15      (encode hippie-plaintext
16             hippie-tree))
```

```
1    <<huffman-trees-txt>>
2    <<encode-txt>>
3    <<encode-symbol>>
4    <<generate-huffman-tree>>
5    (define (successive-merge leaves)
6      (if (null? (cdr leaves))
7          (car leaves)
8          (successive-merge
9           (adjoin-set
10            (make-code-tree (car leaves)
11                            (cadr leaves))
12            (cddr leaves)))))
13    <<hippie-trees>>
14    <<echo>>
15    (load "mattcheck2.scm")
16
17    (echo "Length of unencoded message is" (length hippie-plaintext)
    ↪   "words.")
18    (echo "Length of encoded message is" (length hippie-encoded) "bits")
19    (echo "Unencoded message (newlines added):" (decode hippie-encoded
    ↪   hippie-tree))
```

```
Length of unencoded message is 36 words.
Length of encoded message is 84 bits
Unencoded message (newlines added):
(GET A JOB
SHA NA NA NA NA NA NA NA NA
GET A JOB SHA NA NA NA NA NA NA NA NA
WAH YIP YIP YIP YIP
YIP YIP YIP YIP YIP
SHA BOOM)
```

A fixed-length alphabet would take 3 bits per word, or 108 bits. So the VLE is 77% the size of anything else.

```
1    <<huffman-trees-txt>>
2    <<encode-txt>>
3    <<encode-symbol>>
4    <<generate-huffman-tree>>
5    (define (successive-merge leaves)
```

```
6    (if (null? (cdr leaves))
7        (car leaves)
8        (successive-merge
9         (adjoin-set
10          (make-code-tree (car leaves)
11                          (cadr leaves))
12          (cddr leaves)))))
13   <<hippie-trees>>
14   (define (split-list-every-x list x)
15     (define (rec ll)
16       (if (< (length ll) x)
17           (cons ll '())
18           (cons (list-head ll x)
19                 (rec (list-tail ll x)))))
20     (rec list))
21   (split-list-every-x hippie-encoded 7)
```

$$\begin{array}{ccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 1 & 1 & 0 \\
1 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 0 & 0 \\
1 & 1 & 1 & 1 & 0 & 1 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 & 1 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 1 & 0 & 1 & 1 \\
1 & 0 & 1 & 1 & 0 & 1 & 1 \\
\end{array}$$

50s slander

3.80 Exercise 2.71

3.80.1 Questions

Suppose we have a Huffman tree for an alphabet of n symbols, and that the relative frequencies of the symbols are $1, 2, 4, \ldots, 2^{n-1}$. Sketch the tree for $n = 5$; for $n = 10$. In such a tree (for general n) how many bits are required to encode the most frequent symbol? The least frequent symbol?

3.80.2 Answers

In this type of tree, the most frequent symbol takes 1 bit. The nth symbol takes n bits. If it's the least frequent symbol and n is even, it takes $n - 1$ bits.

3.81 Exercise 2.72: order of growth of `encode`

Let's take a look at `encode-symbol`.

```
;; encode-symbol
(define (encode-symbol symbol tree)
  (define (rec t) ;; rec is O(log n)
    (cond ((and (leaf? t)
                (eq? (symbol-leaf t)
                     symbol)) ; O(1)
           '())
          ((element-of-set? symbol ; O(symbols of left-branch)
```

241

```
 9                                 (symbols (left-branch t)))
10                 (cons 0 ;; O(1)
11                       (rec (left-branch t)))) ;; rec is O(log symbols)
12                 ((element-of-set? symbol ; O(symbols of right-branch)
13                                 (symbols (right-branch t)))
14                 (cons 1 ;; O(1)
15                       (rec (right-branch t)))) ;; rec is O(log symbols)
16                 (else (error "encode-symbol: logic error"))))
17       (if (element-of-set? symbol (symbols tree)) ;; O(symbols)
18           (rec tree) ;; rec is O(log n)
19           (error "encode-symbol: symbol not in tree")))
```

I'm not expecting to get a correct answer, but here's what I'm able to derive. `encode-symbol` uses binary search to find the symbol, which is $\Theta(\log n)$. However, at each step it needs to perform a $\Theta(n)$ search of the left branch. If the symbol isn't found, it will execute a $\Theta(n)$ search of the right branch. So these searches have a varying cost depending on if the tree is biased to the left or the right. So in the worst case of being right-biased, the searches will incur an order of $\Theta(n2)$, while doing the overall binary search of $\Theta(\log n)$. So I think this simplifies to $\Theta(n \log n)$. Time to check my answer...

While I see a couple answers[6] agreeing with mine, I see more saying that `encode-symbol` is $\Theta(1)$ for the most frequent symbol, and $\Theta(n)$ for the average case. Some also say[7] that in the worst case (the least frequent symbol), you may have to visit every non-leaf node for a complexity of $\Theta(n^2)$.

3.82 2.4: Multiple Representations for Abstract Data

We have learned how to abstract "higher" and "lower" parts of a system. Now we will talk about having multiple simultaneous representations of data in the same system. This could be considered "side-by-side" abstraction. The authors tout the

To allow programmers to add and remove representations at will without modifying the original code, the authors suggest using a table lookup system to keep track of representations and their related methods. For example:

Operations	Polar Type	Rectangular Type
real-part	real-part-polar	real-part-rectangular
imag-part	imag-part-polar	imag-part-rectangular
magnitude	magnitude-polar	magnitude-rectangular
angle	angle-polar	angle-rectangular

data-directed programming designing programs to organize their behaviors by a table of operations like the one above.

principle of least commitment abstracting a system so it can work with

---

[6] see the answer of aos here: http://community.schemewiki.org/?sicp-ex-2.72
[7] https://mk12.github.io/sicp/exercise/2/3.html#ex2.72 and others.

many possible representations of data without being rewritten.

The book introduces two methods, (**put** `'operation` `'type` **lambda**) to add such a method to the table, and (**get** `'operation` `'type`) to fetch the saved lambda. I'll use Guile's native hash tables[8] to implement this.

```
(define op-table (make-hash-table))

(define (put operation type f)
  (hash-set! op-table (cons operation type) f))
(define (get operation type)
  (hash-ref op-table (cons operation type)))
```

```
;; basic test of lambda storage
<<op-table>>
<<echo>>
(put 'A 'X (lambda()'AX))
(put 'B 'Y (lambda()'BY))
(put 'C 'Z (lambda()'CZ))

(echo ((get 'A 'X))
      ((get 'B 'Y))
      ((get 'C 'Z)))
```

```
AX BY CZ
```

One method to distinguish these types of items is called **type tagging**. Take this code as an example:

```
(define (attach-tag type-tag contents)
  (cons type-tag contents))

(define (type-tag datum)
  (if (pair? datum)
      (car datum)
      (error "Bad tagged datum:
              TYPE-TAG" datum)))

(define (contents datum)
  (if (pair? datum)
      (cdr datum)
      (error "Bad tagged datum:
              CONTENTS" datum)))
```

3.83 Exercise 2.73: Data-driven symbolic differentiator

2.3.2: Example: Symbolic differentiator described a program that performs symbolic differentiation:

[8]https://www.gnu.org/software/guile/manual/html_node/Hash-Table-Reference.html

```
1   (define (deriv exp var)
2     (cond ((number? exp) 0)
3           ((variable? exp)
4            (if (same-variable? exp var) 1 0))
5           ((sum? exp)
6            (make-sum (deriv (addend exp) var)
7                      (deriv (augend exp) var)))
8           ((product? exp)
9            (make-sum (make-product
10                       (multiplier exp)
11                       (deriv (multiplicand exp) var))
12                      (make-product
13                       (deriv (multiplier exp) var)
14                       (multiplicand exp))))
15          ;; more rules can be added here
16          (else (error "unknown expression type:
17                        DERIV" exp))))
```

We can regard this program as performing a dispatch on the type of the expression to be differentiated. In this situation the "type tag" of the datum is the algebraic operator symbol (such as +) and the operation being performed is deriv. We can transform this program into data-directed style by rewriting the basic derivative procedure as

```
1   (define (deriv exp var)
2     (cond ((number? exp) 0)
3           ((variable? exp) (if (same-variable? exp var) 1 0))
4           (else ((get 'deriv (operator exp))
5                  (operands exp) var))))
6   (define (operator exp) (car exp))
7   (define (operands exp) (cdr exp))
```

3.83.1 Question A

Explain what was done above. Why can't we assimilate the predicates `number?` and `variable?` into the data-directed dispatch?

3.83.2 Answer A

This rewrite removes the predicate-based tests for sums, products, etc, replacing them with the `get`-based lookup system explained earlier.

The call to `get` requires using the `operator` procedures on `exp`, which would fail on a single-item `exp`. So it is necessary to handle the number and variable checks before calling `deriv`.

3.83.3 Questions B and C

- Write the procedures for derivatives of sums and products, and the auxiliary code required to install them in the table used by the program above.

- Choose any additional differentiation rule that you like, such as the one for exponents (Exercise 2.56: Differentiating exponentiation), and install it in this data-directed system.

3.83.4 Answer B & C version 1

Here's my first solution, which is functional, but not at all idiomatic like the textbook's examples of such a system.

```scheme
(define (variable? x) (symbol? x))
(define (same-variable? v1 v2)
  (and (variable? v1) (variable? v2) (eq? v1 v2)))
(define (operator exp) (car exp))
(define (operands exp) (cdr exp))
;; old ^^ vv new
(define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp) (if (same-variable? exp var) 1 0))
        (else ((get 'deriv (operator exp))
               (operands exp) var))))
;; vv old
(define (addend s) (car s)) ;; these and similar procedures
(define (augend s) (cadr s));; modified for 2-item lists of operands
(define (multiplier p) (car p))
(define (multiplicand p) (cadr p))
(define (make-sum a1 a2)
  (cond ((=number? a1 0) a2)
        ((=number? a2 0) a1)
        ((and (number? a1) (number? a2))
         (+ a1 a2))
        (else (list '+ a1 a2))))
(define (=number? exp num)
  (and (number? exp) (= exp num)))
(define (make-product m1 m2)
  (cond ((or (=number? m1 0) (=number? m2 0)) 0)
        ((=number? m1 1) m2)
        ((=number? m2 1) m1)
        ((and (number? m1) (number? m2)) (* m1 m2))
        (else (list '* m1 m2))))
(define (base e) (car e))
(define (exponent e) (cadr e))
(define (make-exponentiation b e)
  (cond ((=number? b 1) 1)
        ((=number? e 0) 1)
        ((=number? e 1) b)
        ((and (number? b) (number? e)) (expt b e))
        (else (list '** b e))))
;; new table-entered functions
```

245

```
40  (put 'deriv '+
41       (lambda (ops var)
42         (make-sum (deriv (addend ops) var)
43                   (deriv (augend ops) var))))
44  (put 'deriv '*
45       (lambda (ops var)
46         (make-sum (make-product
47                     (multiplier ops)
48                     (deriv (multiplicand ops) var))
49                   (make-product
50                     (deriv (multiplier ops) var)
51                     (multiplicand ops)))))
52  (put 'deriv '**
53       (lambda (ops var)
54           (make-product
55            (make-product
56             (exponent ops)
57             (make-exponentiation (base ops)
58                                  (make-sum (exponent ops) -1)))
59            (deriv (base ops) var))))
```

```
1   <<op-table>>
2   <<deriv-datadriven>>
3   (load "mattcheck2.scm")
4   (mattcheck "deriv: basic addition"
5              (deriv '(+ x 3) 'x)
6              1)
7   (mattcheck "deriv: basic multiplication"
8              (deriv '(* x y) 'x)
9              'y)
10
11  (mattcheck "deriv: exponentiation 1"
12             (deriv (make-exponentiation 'a 5) 'a)
13             '(* 5 (** a 4)))
14  (mattcheck "deriv: exponentiation 2"
15             (deriv (make-exponentiation 'a 'b) 'a)
16             '(* b (** a (+ b -1)))))
```

```
SUCCEED at deriv: basic addition
SUCCEED at deriv: basic multiplication
SUCCEED at deriv: exponentiation 1
SUCCEED at deriv: exponentiation 2
```

3.83.5   Answer B & C version 2

```
1   (define (deriv exp var)
2     (define (variable? x) (symbol? x))
```

```
3    (define (same-variable? v1 v2)
4      (and (variable? v1) (variable? v2) (eq? v1 v2)))
5    (define (operator exp) (type-tag exp)) ;; use type tags
6    (define (operands exp) (contents exp))

8    (cond ((number? exp) 0)
9          ((variable? exp) (if (same-variable? exp var) 1 0))
10         (else ((get 'deriv (operator exp))
11                (operands exp) var))))

13 (define (install-deriv-package choice)
14   ;;shared procedures
15   (define (=number? exp num)
16     (and (number? exp) (= exp num)))
17   (define (make-sum a1 a2)
18     (cond ((=number? a1 0) a2)
19           ((=number? a2 0) a1)
20           ((and (number? a1) (number? a2))
21            (+ a1 a2))
22           (else (attach-tag '+ (list a1 a2)))))
23   (define (make-product m1 m2)
24     (cond ((or (=number? m1 0) (=number? m2 0)) 0)
25           ((=number? m1 1) m2)
26           ((=number? m2 1) m1)
27           ((and (number? m1) (number? m2)) (* m1 m2))
28           (else (attach-tag '* (list m1 m2)))))
29   (define (make-exponentiation b e)
30     (cond ((=number? b 1) 1)
31           ((=number? e 0) 1)
32           ((=number? e 1) b)
33           ((and (number? b) (number? e)) (expt b e))
34           (else (attach-tag '** (list b e)))))

36   ;; packages
37   (define (install-sum-package)
38     (define (addend s) (car s))
39     (define (augend s) (cadr s))
40     (put 'deriv '+
41          (lambda (ops var)
42            (make-sum (deriv (addend ops) var)
43                      (deriv (augend ops) var)))))

45   (define (install-mult-package)
46     (define (multiplier p) (car p))
47     (define (multiplicand p) (cadr p))
48     (put 'deriv '*
49          (lambda (ops var)
50            (make-sum (make-product
```

```
51                         (multiplier ops)
52                         (deriv (multiplicand ops) var))
53                      (make-product
54                         (deriv (multiplier ops) var)
55                         (multiplicand ops))))))
56    (define (install-exp-package)
57      (define (base e) (car e))
58      (define (exponent e) (cadr e))
59      (put 'deriv '**
60           (lambda (ops var)
61             (make-product
62              (make-product
63               (exponent ops)
64               (make-exponentiation (base ops)
65                                    (make-sum (exponent ops)
66                                              -1)))
67              (deriv (base ops) var)))))
68
69    ;; body
70    (cond ((eqv? choice 'sum)
71           (install-sum-package))
72          ((eqv? choice 'mult)
73           (install-mult-package))
74          ((eqv? choice 'exp)
75           (install-exp-package))))
```

```
1   <<op-table>>
2   <<type-tagging>>
3   <<deriv-datadriven2>>
4   (load "mattcheck2.scm")
5   (install-deriv-package 'sum)
6   (install-deriv-package 'mult)
7   (install-deriv-package 'exp)
8   (mattcheck "deriv: basic addition"
9              (deriv '(+ x 3) 'x)
10             1)
11  (mattcheck "deriv: basic multiplication"
12             (deriv '(* x y) 'x)
13             'y)
14
15  (mattcheck "deriv: exponentiation 1"
16             (deriv '(** a 5) 'a) ;; no more external access
17             '(* 5 (** a 4)))
18  (mattcheck "deriv: exponentiation 2"
19             (deriv '(** a b) 'a)
20             '(* b (** a (+ b -1))))
```

```
SUCCEED at deriv: basic addition
```

```
SUCCEED at deriv: basic multiplication
SUCCEED at deriv: exponentiation 1
SUCCEED at deriv: exponentiation 2
```

It's notable that Scheme doesn't let you access the sub-definitions inside definitions, so I had to change my tests above. It makes me miss the dot notation that future languages would use for accessing children of objects. It also makes debugging processes less informative, not showing entries in the `,␣ trace foo` system call.

3.83.6   Question D

In this simple algebraic manipulator the type of an expression is the algebraic operator that binds it together. Suppose, however, we indexed the procedures in the opposite way, so that the dispatch line in `deriv` looked like

```
1  ((get (operator exp) 'deriv)
2   (operands exp) var)
```

What corresponding changes to the derivative system are required?

3.83.7   Answer D

Besides the change given in the example above, all the `put` calls would need their arguments switched.

3.84   Exercise 2.74: Multi-division record storage

3.84.1   Questions ABC

Insatiable Enterprises, Inc., is a highly decentralized conglomerate company consisting of a large number of independent divisions located all over the world. The company's computer facilities have just been interconnected by means of a clever network-interfacing scheme that makes the entire network appear to any user to be a single computer. Insatiable's president, in her first attempt to exploit the ability of the network to extract administrative information from division files, is dismayed to discover that, although all the division files have been implemented as data structures in Scheme, the particular data structure used varies from division to division. A meeting of division managers is hastily called to search for a strategy to integrate the files that will satisfy headquarters' needs while preserving the existing autonomy of the divisions.

Show how such a strategy can be implemented with data-directed programming. As an example, suppose that each division's personnel records consist of a single file, which contains a set of records keyed on employees' names. The structure of the set varies from division to division. Furthermore, each employee's record is itself a set (structured differently from division to division) that contains information keyed under identifiers such as `address` and `salary`. In particular:

1. Implement for headquarters a `get-record` procedure that retrieves a specified employee's record from a specified personnel file. The procedure should be applicable to any division's file. Explain how the individual divisions' files should be structured. In particular, what type information must be supplied?

2. Implement for headquarters a `get-salary` procedure that returns the salary information from a given employee's record from any division's personnel file. How should the record be structured in order to make this operation work?

3. Implement for headquarters a `find-employee-record` procedure. This should search all the divisions' files for the record of a given employee and return the record. Assume that this procedure takes as arguments an employee's name and a list of all the divisions' files.

3.84.2   Answers ABC

Let's talk assumptions:

- Assume every division has one file

- Assume HQ wants a consistent schema returned

```
1   <<make-tree>>
2   <<make-record>>
3   <<op-table>>
4   (define (find x ll)
5     (define (iter ll) ;; aka SRFI-1 'find'
6       (cond ((null? ll)
7              #f)
8             ((equal? id (employee-id-east (car ll)))
9              (car ll))
10            (else (iter (cdr ll))))))
11    (iter ll))
12
13  ;; East: records as named cdr aliases
14  (define (install-east-package)
15    (define (make-employee-east id salary)
16      (cons id (/ salary 1000))) ;; East keeps salary in thousands
17    (define (employee-id-east record)
18      (car record))
19    (define (employee-salary-east record)
20      (* (cdr record) 1000))
21    (define East-Employee-List
22      (list (make-employee-east 101 37000) ; bob
23            (make-employee-east 102 48000) ; linda
24            (make-employee-east 103 29000) ; john
25            ))
```

```scheme
26    (define (find-employee-east id)
27      (find id East-Employee-List))
28    (put 'employee-record 'East
29         (lambda (id)
30           (let ((e (find-employee-east id)))
31             (if e
32                 (list (list 'id id)
33                       (list (employee-salary-east e)))
34                 #f))))
35    (put 'employee-salary 'East
36         (lambda(id)
37           (employee-salary-east (find-employee-east id)))))

38
39  ;; records as lookup trees (Ex 2.66)
40  (define (install-west-package)
41    (define (maketree . args)
42      (apply list->tree args))
43    (define (make-employee-west id name salary)
44      (make-record 'id
45                   (maketree (make-record 'name name)
46                             (make-record 'salary salary))))
47    (define West-Record-File
48      (maketree (make-employee-west 201 "James" 55000)
49                (make-employee-west 202 "Joy" 45000)
50                (make-employee-west 203 "Richard" 73000)))
51    (define (employee-west id)
52      (lookup id West-Record-File))
53    (define (employee-attrib-west attrib record)
54      (lookup attrib record))
55    (put 'employee-record 'West
56         (lambda(id)
57           (let ((e (employee-west id)))
58             (if e
59                 (list (list 'id id)
60                       (list 'name (employee-attrib-west 'name e))
61                       (list 'salary (employee-attrib-west 'salary e)))
62                 #f))))
63    (put 'employee-salary 'West
64         (lambda(id)
65           (employee-attrib-west 'salary
66                                 (employee-west id)))))

67
68  ;; South: records as lambdas
69  (define (install-south-package)
70    (define (make-employee-south id name salary)
71      (lambda(value)
72        (cond ((eqv? 'id) id)
73              ((eqv? 'name) name)
```

```scheme
                ((eqv? 'salary) (* 1000
                                   (/ salary 1000))))))
  (define South-Employee-File
    (list (make-employee-south 301 "Jacquelyn" 49000)
          (make-employee-south 302 "Mazie" 66000)))
  (define (find-employee-south id)
    (define (iter ll)
      (cond ((null? ll)
             #f)
            ((equal? id ((car ll) 'id))
             (car ll))
            (else (iter (cdr ll)))))
    (iter South-Employee-File))

  (put 'employee-record 'West
       (lambda(id)
         (let ((e (find-employee-south id)))
           (if e
               (list (list 'id id)
                     (list 'name (e 'id))
                     (list 'salary (e 'salary)))
               #f))))
  (put 'employee-salary 'South
       (lambda(id)
         ((find-employee-west id) 'salary))))

;; HQ record fetcher
(define (get-record employee-id division)
  ((get 'employee-record division)
   employee-id))

;; HQ's salary fetcher
(define (get-salary employee-id division)
  ((get 'employee-salary division)
   employee-id))

;; HQ's employee finder
;; problem asks for "list of files" but since we're working with
;; differently-structured files it must need the procs as well
;; and there's one file per division
(define (find-employee-record id list-of-divisions)
  (if (null? list-of-divisions)
      #f
      (let ((result (get-record id (car list-of-divisions))))
        (if result
            result
            (find-employee-record (cdr list-of-divisions))))))
```

In retrospect I should have designed it so each file had a type tag for the division, methods took a file, and dispatch was made on the file's tag. This

makes more sense in a real-world context.

3.84.3 Question D

When Insatiable takes over a new company, what changes must be made in order to incorporate the new personnel information into the central system?

3.84.4 Answer D

The new division must provide appropriate methods for `get-record` and `get-⏎salary` to use.

3.85 2.4.3: Data-directed and message-passing programs

Altogether the book has introduced 3 ways to organize functions and data types:

1. Generic operations with explicit dispatch: functions which each include **case** statements to dispatch depending on type of data provided. Bulk of the book so far.

2. Data-directed style: given a tagged object and a desired operation, lookup what function to call.

3. Message-passing style: package operations with their appropriate data objects, then fetch those operations from the object.

3.86 Exercise 2.75: Message-passing styled imaginary numbers

3.86.1 Question

Implement the constructor `make-from-mag-ang` in message-passing style. This procedure should be analogous to the `make-from-real-imag` procedure given above.

3.86.2 Answer

```
1  (define (make-from-mag-ang r a)
2    (define (dispatch op)
3      (cond ((eq? op 'magnitude) r)
4            ((eq? op 'angle) a)
5            ((eq? op 'real-part)
6             (* r (cos a)))
7            ((eq? op 'imag-part)
8             (* r (sin a)))
9            (else (error "Unknown op: MAKE-FROM-MAG-ANG" op))))
10
11   dispatch)
```

3.87 Exercise 2.76: comparing generic operation solutions

3.87.1 Question

As a large system with generic operations evolves, new types of data objects or new operations may be needed. For each of the three strategies—generic operations with explicit dispatch, data-directed style, and message-passing-style—describe the changes that must be made to a system in order to add new types or new operations. Which organization would be most appropriate for a system in which new types must often be added? Which would be most appropriate for a system in which new operations must often be added?

3.87.2 Answer

The most important part of this is where code *doesn't* need to be changed.

Generic operations with explicit dispatch let you add new functions without changing existing code.

Message-passing style let you add new data types without modifying existing code.

Data-directed table-lookups let you add new data types *and* functions without modifying existing code.

Let's be clear: this does not really save you much code, only the cognitive load imposed when you have to enter old code to modify it. Let's say T is how many data types you have, O is how many desired operations you have. All of these are $O \times T$, the key is where the code is being stored and what it effects.

This issue of one change requiring traveling across your code to change dozens of definitions is described as the Expression Problem[9]. Data-directed table lookups appear to solve the problem, as discussed by torinmr in this Scheme wiki solution[10].

> "Old code should not stop working when you add new code." -Brian Harvey

Also of note is that this kind of decision-making between generic operations and generic datatypes is commonplace in modern OO programming. I wonder what the old heads from this time period think about OO as it stands today.

3.88 Exercise 2.77: generic arithmetic package

3.88.1 Question

Louis Reasoner tries to evaluate the expression (`magnitude` `z`) where `z` is the object shown in Figure 2.24. To his surprise, instead of the answer 5 he gets

---

[9]http://c2.com/cgi/wiki?ExpressionProblem
[10]http://community.schemewiki.org/?sicp-ex-2.76

an error message from `apply-generic`, saying there is no method for the opera-tion `magnitude` on the types (`complex`). He shows this interaction to Alyssa P. Hacker, who says "The problem is that the complex-number selectors were never defined for `complex` numbers, just for `polar` and `rectangular` numbers. All you have to do to make this work is add the following to the `complex` package":

```
1  (put 'real-part '(complex) real-part)
2  (put 'imag-part '(complex) imag-part)
3  (put 'magnitude '(complex) magnitude)
4  (put 'angle '(complex) angle)
```

Describe in detail why this works. As an example, trace through all the procedures called in evaluating the expression (`magnitude z`) where `z` is the object shown in Figure 2.24. In particular, how many times is `apply-generic` invoked? What procedure is dispatched to in each case?

3.88.2 Answer

Surprised to find I never actually imported `apply-generic`.

```
1  <<op-table>>
2  (define (apply-generic op . args)
3    (let ((type-tags (map type-tag args)))
4      (let ((proc (get op type-tags)))
5        (if proc
6            (apply proc (map contents args))
7            (error
8              "No method for these types: APPLY-GENERIC"
9              (list op type-tags))))))
```

```
1  (define (square x) (* x 2))
2  (define (add x y) (apply-generic 'add x y))
3  (define (sub x y) (apply-generic 'sub x y))
4  (define (mul x y) (apply-generic 'mul x y))
5  (define (div x y) (apply-generic 'div x y))
6  (define (real-part z) (apply-generic 'real-part z))
7  (define (imag-part z) (apply-generic 'imag-part z))
8  (define (magnitude z) (apply-generic 'magnitude z))
9  (define (angle z) (apply-generic 'angle z))
10
11  (define (install-rational-package)
12    ;; internal procedures
13    (define (numer x) (car x))
14    (define (denom x) (cdr x))
15    (define (make-rat n d)
16      (if (or (inexact? n)
17              (inexact? d))
```

```scheme
            (cons n d)
            (let ((g (gcd n d)))
              (cons (/ n g) (/ d g)))))
    (define (add-rat x y)
      (make-rat (+ (* (numer x) (denom y))
                   (* (numer y) (denom x)))
                (* (denom x) (denom y))))
    (define (sub-rat x y)
      (make-rat (- (* (numer x) (denom y))
                   (* (numer y) (denom x)))
                (* (denom x) (denom y))))
    (define (mul-rat x y)
      (make-rat (* (numer x) (numer y))
                (* (denom x) (denom y))))
    (define (div-rat x y)
      (make-rat (* (numer x) (denom y))
                (* (denom x) (numer y))))
    ;; interface to rest of the system
    (define (tag x) (attach-tag 'rational x))
    (put 'add '(rational rational)
         (lambda (x y) (tag (add-rat x y))))
    (put 'sub '(rational rational)
         (lambda (x y) (tag (sub-rat x y))))
    (put 'mul '(rational rational)
         (lambda (x y) (tag (mul-rat x y))))
    (put 'div '(rational rational)
         (lambda (x y) (tag (div-rat x y))))
    (put 'make 'rational
         (lambda (n d) (tag (make-rat n d))))
    (put 'numer '(rational) numer) ;; my addition
    (put 'denom '(rational) denom)
    'done)

;; for future use
(define (numer r)
  ((get 'numer '(rational))
   (contents r)))
(define (denom r)
  ((get 'denom '(rational))
   (contents r)))

(define (make-rational n d)
  ((get 'make 'rational) n d))

(define (install-rectangular-package)
  ;; internal procedures
  (define (real-part z) (car z))
  (define (imag-part z) (cdr z))
```

```scheme
66    (define (make-from-real-imag x y) (cons x y))
67    (define (magnitude z)
68      (sqrt (+ (square (real-part z))
69               (square (imag-part z)))))
70    (define (angle z)
71      (atan (imag-part z) (real-part z)))
72    (define (make-from-mag-ang r a)
73      (cons (* r (cos a)) (* r (sin a))))
74    ;; interface to the rest of the system
75    (define (tag x) (attach-tag 'rectangular x))
76    (put 'real-part '(rectangular) real-part)
77    (put 'imag-part '(rectangular) imag-part)
78    (put 'magnitude '(rectangular) magnitude)
79    (put 'angle '(rectangular) angle)
80    (put 'make-from-real-imag 'rectangular
81         (lambda (x y) (tag (make-from-real-imag x y))))
82    (put 'make-from-mag-ang 'rectangular
83         (lambda (r a) (tag (make-from-mag-ang r a))))
84    'done)

86  (define (install-polar-package)
87    ;; internal procedures
88    (define (magnitude z) (car z))
89    (define (angle z) (cdr z))
90    (define (make-from-mag-ang r a) (cons r a))
91    (define (real-part z) (* (magnitude z) (cos (angle z))))
92    (define (imag-part z) (* (magnitude z) (sin (angle z))))
93    (define (make-from-real-imag x y)
94      (cons (sqrt (+ (square x) (square y)))
95            (atan y x)))
96    ;; interface to the rest of the system
97    (define (tag x) (attach-tag 'polar x))
98    (put 'real-part '(polar) real-part)
99    (put 'imag-part '(polar) imag-part)
100   (put 'magnitude '(polar) magnitude)
101   (put 'angle '(polar) angle)
102   (put 'make-from-real-imag 'polar
103        (lambda (x y) (tag (make-from-real-imag x y))))
104   (put 'make-from-mag-ang 'polar
105        (lambda (r a) (tag (make-from-mag-ang r a))))
106   'done)

108  (define (install-complex-package)
109    ;; imported procedures from rectangular and polar packages
110    (define (make-from-real-imag x y)
111      ((get 'make-from-real-imag 'rectangular) x y))
112    (define (make-from-mag-ang r a)
113      ((get 'make-from-mag-ang 'polar) r a))
```

257

```scheme
    ;; internal procedures
    (define (add-complex z1 z2)
      (make-from-real-imag (+ (real-part z1) (real-part z2))
                           (+ (imag-part z1) (imag-part z2))))
    (define (sub-complex z1 z2)
      (make-from-real-imag (- (real-part z1) (real-part z2))
                           (- (imag-part z1) (imag-part z2))))
    (define (mul-complex z1 z2)
      (make-from-mag-ang (* (magnitude z1) (magnitude z2))
                         (+ (angle z1) (angle z2))))
    (define (div-complex z1 z2)
      (make-from-mag-ang (/ (magnitude z1) (magnitude z2))
                         (- (angle z1) (angle z2))))
    ;; interface to rest of the system
    (define (tag z) (attach-tag 'complex z))
    (put 'add '(complex complex)
         (lambda (z1 z2) (tag (add-complex z1 z2))))
    (put 'sub '(complex complex)
         (lambda (z1 z2) (tag (sub-complex z1 z2))))
    (put 'mul '(complex complex)
         (lambda (z1 z2) (tag (mul-complex z1 z2))))
    (put 'div '(complex complex)
         (lambda (z1 z2) (tag (div-complex z1 z2))))
    (put 'make-from-real-imag 'complex
         (lambda (x y) (tag (make-from-real-imag x y))))
    (put 'make-from-mag-ang 'complex
         (lambda (r a) (tag (make-from-mag-ang r a))))
    (put 'real-part '(complex) real-part)
    (put 'imag-part '(complex) imag-part)
    (put 'magnitude '(complex) magnitude)
    (put 'angle '(complex) angle))

(define (make-complex-from-real-imag x y)
  ((get 'make-from-real-imag 'complex) x y))
(define (make-complex-from-mag-ang r a)
  ((get 'make-from-mag-ang 'complex) r a))
```

```scheme
(define (make-scheme-number n)
  ((get 'make 'scheme-number) n))
(define (install-scheme-number-package)
  (define (tag x) (attach-tag 'scheme-number x))
  (put 'add '(scheme-number scheme-number)
       (lambda (x y) (tag (+ x y))))
  (put 'sub '(scheme-number scheme-number)
       (lambda (x y) (tag (- x y))))
  (put 'mul '(scheme-number scheme-number)
       (lambda (x y) (tag (* x y))))
  (put 'div '(scheme-number scheme-number)
```

```
12          (lambda (x y) (tag (/ x y))))
13      (put 'make 'scheme-number (lambda (x) (tag x)))
14      'done)
```

First I want to do sanity checks on the system.

```
1   <<type-tagging>>
2   <<generic-arithmetic>>
3   <<scheme-number-package>>
4   <<echo>>
5   (load "mattcheck2.scm")
6
7   (install-rational-package)
8   (install-rectangular-package)
9   (install-polar-package)
10  (install-complex-package)
11  (install-scheme-number-package)
12
13  <<gen-math-tests>>
```

```
5+5 = (scheme-number . 10)
5-5 = (scheme-number . 0)
5*5 = (scheme-number . 25)
5/5 = (scheme-number . 1)
1/2 + 1/2 = (rational 1 . 1)
1/2 - 1/2 = (rational 0 . 1)
1/2 * 1/2 = (rational 1 . 4)
1/2 / 1/2 = (rational 1 . 1)
3+4i + 3+4i = (complex rectangular 6 . 8)
3+4i - 3+4i = (complex rectangular 0 . 0)
3+4i * 3+4i = (complex polar 14.0 . 1.8545904360032244)
3+4i / 3+4i = (complex polar 1.0 . 0.0)
3m+4deg + 3m+4deg = (complex rectangular -3.921861725181672 . -4.540814971847569)
3m+4deg - 3m+4deg = (complex rectangular 0.0 . 0.0)
3m+4deg * 3m+4deg = (complex polar 9 . 8)
3m+4deg / 3m+4deg = (complex polar 1 . 0)
```

Now let's run this:

```
1   ,trace (magnitude (make-complex-from-real-imag 3 4))
```

I've heavily pared down this from the real debug output.

```
(make-complex-from-real-imag 3 4)
|  (get make-from-real-imag complex)
|  #<procedure 1ca8ad0 at <unknown port>:166:4 (x y)>
(_ 3 4)
|  (get make-from-real-imag rectangular)
```

```
|   #<procedure 1ca87e8 at <unknown port>:111:4 (x y)>
|   (_ 3 4)
|   (attach-tag rectangular (3 . 4))
|   (rectangular 3 . 4)
(attach-tag complex (rectangular 3 . 4))
(complex rectangular 3 . 4) <- now fully tagged
(magnitude (complex rectangular 3 . 4))
(apply-generic magnitude (complex rectangular 3 . 4)) <- identifying complex number
|   (map #<procedure type-tag (datum)> ((complex rectangular 3 . 4)))
|   (complex)
|   (get magnitude (complex))
|   #<procedure magnitude (z)>
|   (map #<procedure contents (datum)> ((complex rectangular 3 . 4)))
|   ((rectangular 3 . 4))
(_ #<procedure magnitude (z)> ((rectangular 3 . 4))) <- identifying rectangular number
(magnitude (rectangular 3 . 4))
(apply-generic magnitude (rectangular 3 . 4))
|   (map #<procedure type-tag (datum)> ((rectangular 3 . 4)))
|   (rectangular)
|   (get magnitude (rectangular))
|   #<procedure magnitude (z)>
|   (map #<procedure contents (datum)> ((rectangular 3 . 4)))
|   ((3 . 4))
(_ #<procedure magnitude (z)> ((3 . 4)))
(magnitude (3 . 4)) <- finally applying native operations
|   (square 3)
|   6
|   (square 4)
|   8
3.7416573867739413
```

The procedures in `apply-generic` `'complex` are only proxies to other proce-
dures at the moment. However they could be useful for other procedures that
should need to know very little about the `complex` package.

3.89   Exercise 2.78: native scheme numbers

3.89.1   Question

The internal procedures in the `scheme-number` package are essentially nothing
more than calls to the primitive procedures `+`, `-`, etc. It was not possible to
use the primitives of the language directly because our type-tag system requires
that each data object have a type attached to it. In fact, however, all Lisp
implementations do have a type system, which they use internally. Primitive
predicates such as `symbol?` and `number?` determine whether data objects have
particular types. Modify the definitions of `type-tag`, `contents`, and `attach-tag`
from 2.4.2 so that our generic system takes advantage of Scheme's internal type

system. That is to say, the system should work as before except that ordinary numbers should be represented simply as Scheme numbers rather than as pairs whose `car` is the symbol `scheme-number`.

3.89.2   Answer

```scheme
(define (attach-tag type-tag contents)
  (if (number? contents)
      contents
      (cons type-tag contents)))

(define (type-tag datum)
  (cond ((pair? datum)
         (car datum))
        ((number? datum)
         'scheme-number)
        (else (error "Bad tagged datum:
                TYPE-TAG" datum))))

(define (contents datum)
  (cond ((pair? datum)
         (cdr datum))
        ((number? datum)
         datum)
        (else (error "Bad tagged datum:
                CONTENTS" datum))))
```

```scheme
(define msn make-scheme-number)
(echo "5+5 =" (add (msn 5) (msn 5)))
(echo "5-5 =" (sub (msn 5) (msn 5)))
(echo "5*5 =" (mul (msn 5) (msn 5)))
(echo "5/5 =" (div (msn 5) (msn 5)))
(echo "1/2 + 1/2 =" (add (make-rational 1 2)
                         (make-rational 1 2)))
(echo "1/2 - 1/2 =" (sub (make-rational 1 2)
                         (make-rational 1 2)))
(echo "1/2 * 1/2 =" (mul (make-rational 1 2)
                         (make-rational 1 2)))
(echo "1/2 / 1/2 =" (div (make-rational 1 2)
                         (make-rational 1 2)))
(let ((r (make-complex-from-real-imag 3 4))
      (p (make-complex-from-mag-ang 3 4)))
  (echo "3+4i + 3+4i =" (add r r))
  (echo "3+4i - 3+4i =" (sub r r))
  (echo "3+4i * 3+4i =" (mul r r))
  (echo "3+4i / 3+4i =" (div r r))
  (echo "3m+4deg + 3m+4deg =" (add p p))
```

```
21    (echo "3m+4deg - 3m+4deg =" (sub p p))
22    (echo "3m+4deg * 3m+4deg =" (mul p p))
23    (echo "3m+4deg / 3m+4deg =" (div p p)))
```

```
1    <<sqrt>>
2    <<square>>
3    <<echo>>
4    <<smarter-type-tagging>>
5    <<generic-arithmetic>>
6    <<scheme-number-package>>
7    (load "mattcheck2.scm")
8
9    (install-rational-package)
10   (install-rectangular-package)
11   (install-polar-package)
12   (install-complex-package)
13   (install-scheme-number-package)
14
15   <<gen-math-tests>>
```

```
5+5 = 10
5-5 = 0
5*5 = 25
5/5 = 1
1/2 + 1/2 = (rational 1 . 1)
1/2 - 1/2 = (rational 0 . 1)
1/2 * 1/2 = (rational 1 . 4)
1/2 / 1/2 = (rational 1 . 1)
3+4i + 3+4i = (complex rectangular 6 . 8)
3+4i - 3+4i = (complex rectangular 0 . 0)
3+4i * 3+4i = (complex polar 14.0 . 1.8545904360032244)
3+4i / 3+4i = (complex polar 1.0 . 0.0)
3m+4deg + 3m+4deg = (complex rectangular -3.921861725181672 . -4.540814971847569)
3m+4deg - 3m+4deg = (complex rectangular 0.0 . 0.0)
3m+4deg * 3m+4deg = (complex polar 9 . 8)
3m+4deg / 3m+4deg = (complex polar 1 . 0)
```

Confirmed by debugging that these calls aren't adding any tags.

3.90   Exercise 2.79: generic equality

3.90.1   Question

Define a generic equality predicate equ? that tests the equality of two numbers, and install it in the generic arithmetic package. This operation should work for ordinary numbers, rational numbers, and complex numbers.

3.90.2 Answer

```
1  (define (equ? x y) (apply-generic 'equ? x y))
2  (define (install-equ?)
3    (put 'equ? '(scheme-number scheme-number)
4         (lambda(x y) (= x y)))
5    (put 'equ? '(complex complex)
6         (lambda(x y)
7           (let ((t1 (type-tag x))
8                 (t2 (type-tag y)))
9             (cond ((and (eqv? t1 'polar)
10                         (eqv? t2 'polar))
11                    (and (= (apply-generic 'magnitude x)
12                            (apply-generic 'magnitude y))
13                         (= (apply-generic 'angle x)
14                            (apply-generic 'angle y))))
15                   (else
16                    (and (= (apply-generic 'real-part x)
17                            (apply-generic 'real-part y))
18                         (= (apply-generic 'imag-part x)
19                            (apply-generic 'imag-part y))))))))
20    (put 'equ? '(rational rational)
21         (lambda(x y)
22           (and (= ((get 'numer '(rational)) x)
23                   ((get 'numer '(rational)) y))
24                (= ((get 'denom '(rational)) x)
25                   ((get 'denom '(rational)) y)))))
26    (put 'equ? '(real real)
27         (lambda(x y) (= x y))))
```

```
1  <<sqrt>>
2  <<square>>
3  <<echo>>
4  <<smarter-type-tagging>>
5  <<generic-arithmetic>>
6  <<scheme-number-package>>
7  <<equ?>>
8  (load "mattcheck2.scm")
9
10 (install-rational-package)
11 (install-rectangular-package)
12 (install-polar-package)
13 (install-complex-package)
14 (install-scheme-number-package)
15 (install-equ?)
16
17 <<gen-math-tests>>
18
```

263

```
19   (mattcheck "basic equ? true" (equ? 5 5) #t)
20   (mattcheck "basic equ? false" (equ? 4 5) #f)
21   (mattcheck "rect equ? true" (equ? (make-complex-from-real-imag 2 3)
22                                      (make-complex-from-real-imag 2 3)) #t)
23   (mattcheck "rect equ? false" (equ? (make-complex-from-real-imag 2 3)
24                                       (make-complex-from-real-imag 2 5)) #f)
25   (mattcheck "polar equ? true" (equ? (make-complex-from-mag-ang 2 3)
26                                       (make-complex-from-mag-ang 2 3)) #t)
27   (mattcheck "polar equ? false" (equ? (make-complex-from-mag-ang 2 3)
28                                        (make-complex-from-mag-ang 2 5)) #f)
29   (mattcheck "rational equ? true" (equ? (make-rational 2 3)
30                                          (make-rational 2 3)) #t)
31   (mattcheck "rational equ? false" (equ? (make-rational 2 3)
32                                           (make-rational 2 5)) #f)
```

```
<gen-math-tests output>
SUCCEED at basic equ? true
SUCCEED at basic equ? false
SUCCEED at rect equ? true
SUCCEED at rect equ? false
SUCCEED at polar equ? true
SUCCEED at polar equ? false
SUCCEED at rational equ? true
SUCCEED at rational equ? false
```

3.91 Exercise 2.80: Generic testing for zero

3.91.1 Question

Define a generic predicate =zero? that tests if its argument is zero, and install
it in the generic arithmetic package. This operation should work for ordinary
numbers, rational numbers, and complex numbers.

3.91.2 Answer

```
1    (define (=zero? x) (apply-generic '=zero? x))
2    (define (install-=zero?)
3      (put '=zero? '(scheme-number)
4           (lambda(x) (= x 0)))
5      (put '=zero? '(complex)
6           (lambda(x)
7             (apply-generic '=zero? x)))
8      (let ((magnitude (get 'magnitude '(polar))))
9        (put '=zero? '(polar)
10             (lambda(x)
11               (= 0 (magnitude x)))))
12     (let ((real-part (get 'real-part '(rectangular)))
13           (imag-part (get 'imag-part '(rectangular))))
```

```
14        (put '=zero? '(rectangular)
15            (lambda(x)
16              (and (= 0 (real-part x))
17                   (= 0 (imag-part x)))))))
18      (put '=zero? '(rational)
19          (lambda(x)
20            (or (= 0 ((get 'numer '(rational)) x))
21                (= 0 ((get 'denom '(rational)) x)))))))
```

```
1    <<sqrt>>
2    <<square>>
3    <<echo>>
4    <<smarter-type-tagging>>
5    <<generic-arithmetic>>
6    <<scheme-number-package>>
7    <<equ?>>
8    <<eqzero?>>
9    (load "mattcheck2.scm")
10
11   (install-rational-package)
12   (install-rectangular-package)
13   (install-polar-package)
14   (install-complex-package)
15   (install-scheme-number-package)
16   (install-equ?)
17   (install-=zero?)
18   <<gen-math-tests>>
19
20   (mattcheck "basic =zero? true" (=zero? 0) #t)
21   (mattcheck "basic =zero? false" (=zero? 5) #f)
22   (mattcheck "rect =zero? true" (=zero? (make-complex-from-real-imag 0 0))
     ↪   #t)
23   (mattcheck "rect =zero? false" (=zero? (make-complex-from-real-imag 2
     ↪   5)) #f)
24   (mattcheck "polar =zero? true" (=zero? (make-complex-from-mag-ang 0 3))
     ↪   #t)
25   (mattcheck "polar =zero? false" (=zero? (make-complex-from-mag-ang 2 0))
     ↪   #f)
26   (mattcheck "rational =zero? true" (=zero? (make-rational 0 3)) #t)
27   (mattcheck "rational =zero? false" (=zero? (make-rational 2 3)) #f)
```

```
<gen-math-tests output>

SUCCEED at basic =zero? true
SUCCEED at basic =zero? false
SUCCEED at rect =zero? true
SUCCEED at rect =zero? false
SUCCEED at polar =zero? true
```

```
SUCCEED at polar =zero? false
SUCCEED at rational =zero? true
SUCCEED at rational =zero? false
```

3.92   2.4.3: Combining Data of Different Types

So far, the generic type system has had all types be completely independent.
However, all of these number systems have relations between each other, specifically they can be converted in this order:

integers rational real complex

Converting types to each other is called **coercion**. With changes to `apply-`
`generic`, the system can automatically find compatible coercions and use them.
For example, $\frac{1}{3} \times 3$ is now possible.

```
1   (define (apply-generic op . args)
2     (let ((type-tags (map type-tag args)))
3       (let ((proc (get op type-tags)))
4         (if proc
5             (apply proc (map contents args))
6             (if (= (length args) 2)
7                 (let ((type1 (car type-tags))
8                       (type2 (cadr type-tags))
9                       (a1 (car args))
10                      (a2 (cadr args)))
11                   (let ((t1->t2
12                          (get-coercion type1
13                                        type2))
14                         (t2->t1
15                          (get-coercion type2
16                                        type1)))
17                     (cond (t1->t2
18                            (apply-generic
19                             op (t1->t2 a1) a2))
20                           (t2->t1
21                            (apply-generic
22                             op a1 (t2->t1 a2)))
23                           (else
24                            (error
25                             "No method for
26                              these types"
27                             (list
28                              op
29                              type-tags))))))
30                 (error
31                  "No method for these types"
32                  (list op type-tags)))))))
```

This assumes `put-coercion` and `get-coercion`.

266

```
1  (define type-table (make-hash-table))
2
3  (define (put-coercion type1 type2 f)
4    (hash-set! type-table (cons type1 type2) f))
5  (define (get-coercion type1 type2)
6    (hash-ref type-table (cons type1 type2)))
```

Authors observe that much work in programs using object-oriented languages is around type coercion.

To write coercion procedures between every type would require n^2 routines. However, by keeping track of the relations between types (similarly to a tree or graph), we can find paths through multiple coercion routines at once. i.e. we can coerce type A to type D by coercion routines A->B, B->C, and C->D.

3.93 Exercise 2.81: Louis' `apply-generic`

Louis Reasoner has noticed that `apply-generic` may try to coerce the arguments to each other's type even if they already have the same type. Therefore, he reasons, we need to put procedures in the coercion table to coerce arguments of each type to their own type. For example, in addition to the `scheme-number-`↵ `>complex` coercion shown above, he would do:

```
1  (define (scheme-number->scheme-number n) n)
2  (define (complex->complex z) z)
3  (put-coercion 'scheme-number
4                'scheme-number
5                scheme-number->scheme-number)
6  (put-coercion 'complex 'complex complex->complex)
```

3.93.1 Question A

With Louis's coercion procedures installed, what happens if `apply-generic` is called with two arguments of type `scheme-number` or two arguments of type ↵ `complex` for an operation that is not found in the table for those types? For example, assume that we've defined a generic exponentiation operation:

```
1  (define (exp x y) (apply-generic 'exp x y))
```

and have put a procedure for exponentiation in the Scheme-number package but not in any other package:

```
1  ;; following added to Scheme-number package
2  (put 'exp '(scheme-number scheme-number)
3       (lambda (x y) (tag (expt x y))))
4       ; using primitive expt
```

What happens if we call `exp` with two complex numbers as arguments?

267

3.93.2 Answer A

I think it will loop infinitely at the first nested `apply-generic` call, seen in the block after `t1->t2`.

3.93.3 Question B

Is Louis correct that something had to be done about coercion with arguments of the same type, or does `apply-generic` work correctly as is?

3.93.4 Answer B

He is correct. The predicate which determines whether type coercion is attempted is only a check to if the requested procedure exists. This is a problem if, as seen in the last problem, the data types are compatible but the operation does not exist.

3.93.5 Question C

Modify `apply-generic` so that it doesn't try coercion if the two arguments have the same type.

3.93.6 Answer C

```
1   (define (apply-generic op . args)
2     (let ((type-tags (map type-tag args)))
3       (let ((proc (get op type-tags)))
4         (if proc
5             (apply proc (map contents args))
6             (if (= (length args) 2)
7                 (let ((type1 (car type-tags))
8                       (type2 (cadr type-tags))
9                       (a1 (car args))
10                      (a2 (cadr args)))
11                  (if (equal? type1 type2)
12                   ;      'failgood ; proving correct operation
13                        (error
14                     "No method for this type"
15                     (list op type-tags))
16                    (let ((t1->t2
17                            (get-coercion type1
18                                          type2))
19                          (t2->t1
20                            (get-coercion type2
21                                          type1)))
22                      (cond (t1->t2
23                             (apply-generic
24                              op (t1->t2 a1) a2))
```

268

```
25                            (t2->t1
26                             (apply-generic
27                              op a1 (t2->t1 a2)))
28                            (else
29                             (error
30                              "No coercion method for these types"
31                              (list
32                               op
33                               type-tags)))))))
34                   (error
35                    "No method for these types"
36                    (list op type-tags)))))))
```

```
1    (put-coercion 'scheme-number 'rational
2                  (lambda(num)
3                    (make-rational num 1)))
4    <<eqnumber>>
5    (put-coercion 'rational 'complex
6                  (lambda(num)
7                    (make-complex-from-real-imag
8                     (if (=number? 1 ((get 'denom '(rational)) num)) ; if
                     ↪  rat could be integer
9                         ((get 'numer '(rational)) num)       ; might as
                     ↪  well simplify
10                        num) 0)))
11   (put-coercion 'scheme-number 'complex
12                 (lambda(num)
13                   (make-complex-from-real-imag num 0)))
```

```
1    <<op-table>>
2    <<type-table>>
3    <<sqrt>>
4    <<square>>
5    <<echo>>
6    <<smarter-type-tagging>>
7    <<apply-generic-coerce-fixed>>
8    <<generic-arithmetic>>
9    <<scheme-number-package>>
10   <<equ?>>
11   <<eqzero?>>
12   (load "mattcheck2.scm")
13
14   (install-rational-package)
15   (install-rectangular-package)
16   (install-polar-package)
17   (install-complex-package)
18   (install-scheme-number-package)
```

```
19    (install-equ?)
20    (install-=zero?)
21    <<gen-math-tests>>
22    <<basic-coercions>>
23
24    (mattcheck "good coercion test"
25              (mul (make-rational 8 1)
26                   (make-rational 2 3))
27              (mul 8 (make-rational 2 3)))
28    (mattcheck "bad coercion test"
29              (apply-generic 'foobar 8 9)
30              'failgood)
```

```
SUCCEED at good coercion test
SUCCEED at bad coercion test
```

3.94 Exercise 2.82: `apply-generic` with multiple coerced arguments

3.94.1 Question

Show how to generalize `apply-generic` to handle coercion in the general case of multiple arguments. One strategy is to attempt to coerce all the arguments to the type of the first argument, then to the type of the second argument, and so on. Give an example of a situation where this strategy (and likewise the two-argument version given above) is not sufficiently general. (Hint: Consider the case where there are some suitable mixed-type operations present in the table that will not be tried.)

3.94.2 Answer

These strategies are not sufficient when a path to the correct type will require multiple coercions. Say you want to coerce arguments of types $ABCD$. There are coercions for $A \to B$, $B \to C$, $A \to D$, $C \to A$ but *not* $C \to D$. If fed arguments of types $ABCD$, `apply-generic` will first coerce A to B, then B to C, and then fail to find a coercion from C to D despite a path $C \to A \to D$ existing.

The other scenario is if there is a different type not included in the original set which all types could convert to. For example if coercions to E exist for all $ABCD$, this would not be checked for.

Here's an implementation of the first strategy, which is a mess:

```
1    (use-srfis '(1))
2    ;; Version of apply-generic that look for coercion functions
3    ;; that work for the entire list
4    (define (apply-generic op . args)
5      (define types (map type-tag args))
```

270

```
 6    (define (coercions-for-list ll target)
 7      (map (lambda(item) ; find coercion functions, if identical type
        ↪  return id func
 8            (let ((type (type-tag item)))
 9              (if (equal? type target)
10                  (lambda(x)x)
11                  (get-coercion type target))))
12          args))
13    ;; map-coercions: list of types -> list of tagged numbers.
14    ;; find one type which all items can be coerced to,
15    ;; then do all coercions and return as list
16    (define (map-coercions ll)
17      (if (null? ll) ;; exhausted possibilities
18          (error
19           "No method between all these types"
20           (list op types))
21          (let ((tries (coercions-for-list ll (car ll))))
22            (if (every procedure? tries); if all coercions found, apply
              ↪  them
23                (map (lambda(f x) (f x))
24                     tries args)
25                (map-coercions (cdr ll)))))) ; else keep trying
26    (let ((proc (get op types)))
27      (if proc
28          (apply proc (map contents args))
29          (let ((coerced (map-coercions types)))
30            (apply (get op (map type-tag coerced))
31                   (map contents coerced))))))
```

Testing my code:

```
 1  <<op-table>>
 2  <<type-table>>
 3  <<sqrt>>
 4  <<square>>
 5  <<echo>>
 6  <<smarter-type-tagging>>
 7  <<apply-generic-coerce-varargs-naive>>
 8  <<generic-arithmetic>>
 9  <<scheme-number-package>>
10  <<equ?>>
11  <<eqzero?>>
12  (load "mattcheck2.scm")
13
14  (install-rational-package)
15  (install-rectangular-package)
16  (install-polar-package)
17  (install-complex-package)
18  (install-scheme-number-package)
```

271

```
19   (install-equ?)
20   (install-=zero?)
21   <<gen-math-tests>>
22   <<basic-coercions>>
23
24   (mattcheck "num -> frac coercion test"
25              (mul (make-rational 8 1)
26                   (make-rational 2 3))
27              (mul 8 (make-rational 2 3)))
28   (mattcheck "num -> complex coercion test"
29              (mul (make-complex-from-real-imag 8 0)
30                   (make-complex-from-real-imag 2 3))
31              (mul 8 (make-complex-from-real-imag 2 3)))
```

```
SUCCEED at num -> frac coercion test
SUCCEED at num -> complex coercion test
```

The strategy I implemented was "search the list and find one type which all others coerce to." However, a superior strategy, as implemented by danielpi on GitHub[11] is to iterate through the list's types, attempting every coercion that works, and leaving unmodified every number that doesn't have one. In a linear "tower" structure, the types will raise as far as is necessary – assuming the types are ordered lowest-to-highest.

Here's my attempt at a much simplified version.

```
1    (use-srfis '(1))
2    ;; Version of apply-generic that tries to coerce each variable
3    ;; into each type represebted in args.
4    (define (apply-generic op . args)
5      (define types (map type-tag args))
6      ;; map-coercion: type, list of tagged numbers -> list of tagged numbers
7      ;; try to coerce all items in list to given type. If this isn't
         ↪ possible,
8      ;; leave the item alone. Meant for fold.
9      (define (map-coercion type-to-try items)
10       (map (lambda(item) ; find and apply coercion functions,
11             (let* ((coerce (get-coercion (type-tag item) type-to-try)))
12               (cond ((or (not coerce)
13                          (equal? (type-tag item) type-to-try))
14                      item)
15                     (else (coerce item)))))
16           items))
17
18     (let ((proc (get op types)))
19       (if proc ; is this already doable?
```

[11]starting on line 269 https://github.com/danielpi/SICP-Exercises/blob/46a6b38fd83faae21db239f40d913ff24062d2e6/Racket/Ex%202.82%20Generalized%20apply-generic.rkt#L269

```
20          (apply proc (map contents args))
21          ; if not, try type coercion
22          (let ((coerced (fold map-coercion args types)))
23            (apply (get op (map type-tag coerced))
24                   (map contents coerced))))))))
25   ;; However this won't actually handle multiple arguments
```

```
1   <<op-table>>
2   <<type-table>>
3   <<sqrt>>
4   <<square>>
5   <<echo>>
6   <<smarter-type-tagging>>
7   <<apply-generic-coerce-less-naive>>
8   <<generic-arithmetic>>
9   <<scheme-number-package>>
10  <<equ?>>
11  <<eqzero?>>
12  (load "mattcheck2.scm")
13
14  (install-rational-package)
15  (install-rectangular-package)
16  (install-polar-package)
17  (install-complex-package)
18  (install-scheme-number-package)
19  (install-equ?)
20  (install-=zero?)
21  <<gen-math-tests>>
22  <<basic-coercions>>
23
24  (mattcheck "num -> frac coercion test"
25             (mul (make-rational 8 1)
26                  (make-rational 2 3))
27             (mul 8 (make-rational 2 3)))
28  (mattcheck "num -> complex coercion test"
29             (mul (make-complex-from-real-imag 8 0)
30                  (make-complex-from-real-imag 2 3))
31             (mul 8 (make-complex-from-real-imag 2 3)))
32  (mattcheck "multiple arguments test"
33             (false-if-exception (apply-generic 'add 8 (make-rational 2 3)
34                                                 (make-complex-from-real-im⌋
                                                ↪   ag 2
                                                ↪   3))))
```

```
SUCCEED at num -> frac coercion test
SUCCEED at num -> complex coercion test
FAIL at multiple arguments test
expected: "boolean true"
returned: "boolean false"
```

273

3.94.3 Answer with vararg support

```
1   (use-srfis '(1))
2   ;; Version of apply-generic that tries to coerce each variable
3   ;; into each type represebted in args.
4   (define (apply-generic op . args)
5     (define types (map type-tag args))
6     ;; map-coercion: type, list of tagged numbers -> list of tagged numbers
7     ;; try to coerce all items in list to given type. If this isn't
    ↪  possible,
8     ;; leave the item alone. Meant for fold.
9     (define (map-coercion type-to-try items)
10      (map (lambda(item) ; find and apply coercion functions,
11            (let* ((coerce (get-coercion (type-tag item) type-to-try)))
12              (cond ((or (not coerce)
13                         (equal? (type-tag item) type-to-try))
14                     item)
15                    (else (coerce item)))))
16          items))
17
18    (let ((proc (get op types)))
19      (if proc ; is this already doable?
20          (apply proc (map contents args))
21          ; if not, try type coercion
22          (let ((coerced (fold map-coercion args types)))
23            (if (< (length args) 3)
24                ;; short args
25                (apply (get op (map type-tag coerced))
26                       (map contents coerced))
27                ;; long args
28                (reduce (lambda(elem previous)
29                          (apply-generic op previous elem))
30                        "ERROR" coerced))))))
```

```
1   <<op-table>>
2   <<type-table>>
3   <<sqrt>>
4   <<square>>
5   <<echo>>
6   <<smarter-type-tagging>>
7   <<apply-generic-coerce-varargs>>
8   <<generic-arithmetic>>
9   <<scheme-number-package>>
10  <<equ?>>
11  <<eqzero?>>
12  (load "mattcheck2.scm")
13
14  (install-rational-package)
```

274

```
15   (install-rectangular-package)
16   (install-polar-package)
17   (install-complex-package)
18   (install-scheme-number-package)
19   (install-equ?)
20   (install-=zero?)
21   <<basic-coercions>>
22
23   (mattcheck "multiple arguments test"
24             (apply-generic 'add 8 2 3)
25             13)
26   (mattcheck "multiple arguments test 2"
27             (apply-generic 'add (make-complex-from-real-imag 3 5)
28                                 (make-complex-from-real-imag 3 5)
29                                 (make-complex-from-real-imag 3 5))
30             (cons 'complex
31                   (cons 'rectangular
32                         (cons 9 15))))
33   (mattcheck "multiple arguments coercion test"
34             (false-if-exception
35              (apply-generic 'add 8 (make-rational 2 3)
36              (make-complex-from-real-imag 2 3))))
```

```
<unknown-location>: warning: possibly unbound variable `fold'
<unknown-location>: warning: possibly unbound variable `reduce'
<unknown-location>: warning: possibly unbound variable `mattcheck'
SUCCEED at multiple arguments test
SUCCEED at multiple arguments test 2
FAIL at multiple arguments coercion test
expected: "boolean true"
returned: "boolean false"
```

The second test fails because of the rational being nested within the complex number type. Taking this any further would involve reworking earlier procedures (such as `add-complex`) and likely writing the future `drop` procedure anyway.

So let's move on.

3.95   Exercise 2.83: `raise`

3.95.1   Question

Suppose you are designing a generic arithmetic system for dealing with the tower of types shown in Figure 2.25: integer, rational, real, complex. For each type (except complex), design a procedure that raises objects of that type one level in the tower. Show how to install a generic `raise` operation that will work for each type (except complex).

3.95.2 Answer

I'm again getting the feeling that the authors are asking for a theoretical exercise, but one that I'm not satisfied leaving to theory. First we need a structure to clarify the promotional possibilities of the tower. Then we need coercions for each type which can be used when following the tower.

Something which I haven't defined before but will now is a `'real` type. A real number is a superset of ints and rationals, but isn't fundamentally different from these until a non-zero imaginary part is added. So my first instinct was to make it an untagged type like `'scheme-number` that is detected by `type-tag` with the builtin predicate `real?`. However after thinking about it some more I think I'll just implement it in the type table and have it pass its arguments off to the "real" types.

Real numbers are a superset of integers, rationals, and **irrational** numbers, which are not a part of this system yet. Also it turns out this module expected a "real" number to be a decimal. -Matt

```
1  (define (=number? x y)
2    (and (number? x)
3         (number? y)
4         (= x y)))
5  (define (=int? x y)
6    (and (number? x)
7         (number? y)
8         (integer? x)
9         (integer? y)
10        (= x y)))
```

```
1  (define tower
2    '(scheme-number rational real complex))
3
4  (define (raise x)
5    (let ((t1 (type-tag x)))
6      (let recurse ((ll tower))
7        (let ((t2 (car ll)))
8          (cond ((null? (cdr ll)) #f)
9                ((equal? t1 t2)
10                 ((get-coercion t1 (cadr ll)) (contents x)))
11                (else (recurse (cdr ll)))))))))
12
13  (put-coercion 'scheme-number 'rational
14                (lambda(num)
15                  (make-rational num 1)))
16  <<eqnumber>>
17  (put-coercion 'rational 'real
18                (lambda(num)
19                  (make-real
```

276

```
20                    (exact->inexact (/ ((get 'numer '(rational)) num)
21                                       ((get 'denom '(rational)) num))))))
22  (put-coercion 'real 'complex
23                  (lambda(num)
24                    (make-complex-from-real-imag
25                      num 0)))
```

```
1   (define (install-real-package)
2     ;; interface to rest of the system
3     (define (tag x) (attach-tag 'real x))
4     (put 'add '(real real)
5          (lambda (x y) (tag (add x y))))
6     (put 'sub '(real real)
7          (lambda (x y) (tag (sub x y))))
8     (put 'mul '(real real)
9          (lambda (x y) (tag (mul x y))))
10    (put 'div '(real real)
11         (lambda (x y) (tag (div x y))))
12    (put 'make 'real
13         (lambda (r) (tag r)))
14    'done)
15
16  (define (make-real r)
17    ((get 'make 'real) r))
```

```
1   <<op-table>>
2   <<type-table>>
3   <<sqrt>>
4   <<square>>
5   <<echo>>
6   <<smarter-type-tagging>>
7   <<attach-tag-fixed>>  ;; time travel due to numer/denom fixes
8   <<apply-generic-coerce-less-naive>>
9   <<generic-arithmetic>>
10  <<scheme-number-package>>
11  <<real-package>>
12  <<equ?>>
13  <<eqzero?>>
14  <<real-package>>
15  <<basic-tower-coercions>>
16
17  (install-rational-package)
18  (install-rectangular-package)
19  (install-polar-package)
20  (install-complex-package)
21  (install-scheme-number-package)
22  (install-real-package)
```

277

```
23   (install-equ?)
24   (install-=zero?)
25   <<gen-math-tests>>
26
27   (let recurse ((x 42))
28     (let ((x2 (raise x)))
29       (if (eqv? #f x2)
30           (echo "Done")
31           (begin (echo x2)
32                  (recurse x2)))))
```

```
<unknown-location>: warning: possibly unbound variable `fold'
5+5 = 10
5-5 = 0
5*5 = 25
5/5 = 1
1/2 + 1/2 = (rational 1 . 1)
1/2 - 1/2 = (rational 0 . 1)
1/2 * 1/2 = (rational 1 . 4)
1/2 / 1/2 = (rational 1 . 1)
3+4i + 3+4i = (complex rectangular 6 . 8)
3+4i - 3+4i = (complex rectangular 0 . 0)
3+4i * 3+4i = (complex polar 14.0 . 1.8545904360032244)
3+4i / 3+4i = (complex polar 1.0 . 0.0)
3m+4deg + 3m+4deg = (complex rectangular -3.921861725181672 . -4.540814971847569)
3m+4deg - 3m+4deg = (complex rectangular 0.0 . 0.0)
3m+4deg * 3m+4deg = (complex polar 9 . 8)
3m+4deg / 3m+4deg = (complex polar 1 . 0)
(rational 42 . 1)
(real . 42.0)
(complex rectangular (real . 42.0) . 0)
Done
```

3.96 Exercise 2.84: `append-generic` with `raise`

3.96.1 Question

Using the `raise` operation of Exercise 2.83, modify the `apply-generic` procedure so that it coerces its arguments to have the same type by the method of successive raising, as discussed in this section. You will need to devise a way to test which of two types is higher in the tower. Do this in a manner that is "compatible" with the rest of the system and will not lead to problems in adding new levels to the tower.

3.96.2 Answer

First some (unnecessarily general) ways to find items in a list.

```scheme
(use-srfis '(1))
;; list, items to find -> list of indices
;; example: (list-indices '(c b a) a b c z)
;;          (2 1 0 #f)
;; takes a list and items to find in the list
;; returns the index of each or #f
(define (list-indices lst . items)
  ;; folder: item, ongoing list of search results
  ;;              -> list of search results
  ;; List of results is a list of #f for unfound items
  ;; and indices for found ones. Take an item to be
  ;; searched for in lst and prior search results. Check
  ;; for each unfound item and if now found, mark index.
  (define (folder x counters)
    (cons (1+ (car counters))
          (let rec ((il items)
                    (cl (cdr counters)))
            (cons (if (and (equal? (car cl) #f)
                           (equal? x (car il)))
                      (car counters)
                      (car cl))
                  (if (null? (cdr il))
                      '()
                      (rec (cdr il) (cdr cl)))))))
  (if (= 1 (length items))
      ;; list-index is good enough
      (cons (list-index (lambda(x)
                          (equal? x (car items)))
                        lst)
            '())
      (cdr (fold folder (cons 0
                             (make-list (length items) #f))
                 lst))))
(define (max-index lst)
  (define (folder x result)
    (let ((index (car result))
          (max-i (cadr result))
          (max-value (caddr result)))
      (cons (1+ index)
            (if (> x max-value)
                (list (1+ index)
                      x)
                (list max-i
                      max-value)))))

  (cadr (fold folder (list -1 0 0)
              lst)))
```

```
49   ;; My first definition for max-index was this under the
50   ;; assumption it would be faster, because the former
51   ;; makes variables for each fold. But they benchmarked
52   ;; identically.
53   ;(define (max-index lst)
54   ;  (define index car)
55   ;  (define max-i cadr)
56   ;  (define max-value caddr)
57   ;  (define (folder x result)
58   ;    (cons (1+ (index result))
59   ;          (if (> x (max-value result))
60   ;              (list (1+ (index result))
61   ;                    x)
62   ;              (list (max-i result)
63   ;                    (max-value result)))))
64   ;
65   ;  (max-i (fold folder (list -1 0 0)
66   ;               lst)))
```

The new apply-generic and helper functions.

```
1    ;; type, type -> bool
2    ;; Return true if type a is the lowest, false otherwise.
3    (define (is-earlier-in-tower? a b)
4      (let ((result (list-indices tower a b)))
5        (if (any (lambda(x) (eq? x #f)) result)
6            (error "type missing from tower!" result)
7            (apply < result))))
8    ;; types -> type
9    ;; Take types to check on tower. Finds the
10   ;; highest-ranked and returns it.
11   (define (latest-in-tower . args)
12     (let* ((types (if (and (= 1 (length args))
13                            (list? (car args)))
14                       (car args)
15                       args))
16            (result (apply list-indices
17                           tower types)))
18       (if (any (lambda(x) (eq? x #f)) result)
19           (error "type missing from tower!"
20                  "args" args "types" types "result" result)
21           (list-ref types (max-index result)))))
```

```
1    (use-srfis '(1))
2    ;; Version of apply-generic that tries raising
3    ;; its arguments.
4    (define (apply-generic op . args)
5      (define types (map type-tag args))
```

```
6    (define (map-raise highest-type item)
7      (if (equal? (type-tag item) highest-type)
8          item
9          (let ((raised (raise item)))
10           (if (not raised)
11               (error "map-raise: could not raise:" item highest-type)
12               (map-raise highest-type raised)))))
13
14   (let ((proc (get op types)))
15     (if proc ; is this already doable?
16         (apply proc (map contents args))
17                                    ; if not, try type coercion
18
19         (let* ((highest-type (apply latest-in-tower types))
20                (map-raise-prime (lambda(item)(map-raise highest-type
                 ↪  item)))
21                (raised (map map-raise-prime args)))
22           (if (< (length args) 3)
23               ;; short args
24               (apply (get op (map type-tag raised))
25                      (map contents raised))
26               ;; long args
27               (reduce (lambda(elem previous)
28                         (apply-generic op previous elem))
29                       "ERROR" raised))))))
```

I discovered that the prior method for avoiding tagging scheme-numbers also keeps you from tagging other things. Here's one that avoids the problem:

```
1    (define (attach-tag type-tag contents)
2      (if (eqv? type-tag 'scheme-number)
3          contents
4          (cons type-tag contents)))
```

```
1    (load "mattcheck2.scm")
2    <<op-table>>
3    <<type-table>>
4    <<sqrt>>
5    <<square>>
6    <<echo>>
7    <<smarter-type-tagging>>
8    <<attach-tag-fixed>>
9    <<list-indices>>
10   <<tower-predicates>>
11   <<apply-generic-raise>>
12   <<generic-arithmetic>>
13   <<scheme-number-package>>
14   <<real-package>>
```

```
15  <<equ?>>
16  <<eqzero?>>
17  <<real-package>>
18  <<basic-tower-coercions>>
19
20  (install-rational-package)
21  (install-rectangular-package)
22  (install-polar-package)
23  (install-complex-package)
24  (install-scheme-number-package)
25  (install-real-package)
26  (install-equ?)
27  (install-=zero?)
28  <<gen-math-tests>>
29
30  (mattcheck "is-earlier-in-tower? true"
31            (is-earlier-in-tower? 'scheme-number 'real))
32  (mattcheck "is-earlier-in-tower? false"
33            #f (is-earlier-in-tower? 'complex 'scheme-number))
34  (mattcheck "max-index"
35            (max-index (list 1 3 5 4 2 0))
36            2)
37  (mattcheck "latest-in-tower"
38            (apply latest-in-tower tower)
39            'complex)
40  (mattcheck "latest-in-tower reverse"
41            (apply latest-in-tower (reverse tower))
42            'complex)
43  (mattcheck "latest-in-tower 2 args"
44            (apply latest-in-tower '(scheme-number real))
45            'real)
46  (mattcheck "latest-in-tower 1 arg"
47            (apply latest-in-tower '(real))
48            'real)
49  (mattcheck "raise success"
50            (raise 78)
51            (make-rational 78 1))
52  (mattcheck "raise fail"
53            (raise (make-complex-from-mag-ang 78 64))
54            #f)
55  (mattcheck "apply-generic 1"
56            (add 1 (make-rational 3 5))
57            (make-rational 8 5))
58  (mattcheck "apply-generic 2"
59            (add (make-real 2)
60                 (make-complex-from-mag-ang 3 5))
61            (make-complex-from-real-imag 2.850986556389679
             ↪  -2.8767728239894153))
```

5+5 = 10

```
5-5 = 0
5*5 = 25
5/5 = 1
1/2 + 1/2 = (rational 1 . 1)
1/2 - 1/2 = (rational 0 . 1)
1/2 * 1/2 = (rational 1 . 4)
1/2 / 1/2 = (rational 1 . 1)
3+4i + 3+4i = (complex rectangular 6 . 8)
3+4i - 3+4i = (complex rectangular 0 . 0)
3+4i * 3+4i = (complex polar 14.0 . 1.8545904360032244)
3+4i / 3+4i = (complex polar 1.0 . 0.0)
3m+4deg + 3m+4deg = (complex rectangular -3.921861725181672 . -4.540814971847569)
3m+4deg - 3m+4deg = (complex rectangular 0.0 . 0.0)
3m+4deg * 3m+4deg = (complex polar 9 . 8)
3m+4deg / 3m+4deg = (complex polar 1 . 0)
SUCCEED at is-earlier-in-tower? true
SUCCEED at is-earlier-in-tower? false
SUCCEED at max-index
SUCCEED at latest-in-tower
SUCCEED at latest-in-tower reverse
SUCCEED at latest-in-tower 2 args
SUCCEED at latest-in-tower 1 arg
SUCCEED at raise success
SUCCEED at raise fail
SUCCEED at apply-generic 1
SUCCEED at apply-generic 2
```

In this implementation, In order to add a new type, all that will be needed is to replace the tower variable (the horrors of setq...) and add a coercion to the respective type.

My first attempt at this exercise had a problem from 'real and above, where numbers passed to math primitives would fail, because of receiving numbers tagged 'real.

```
1   (add (make-real 2)
2        (make-complex-from-mag-ang 3 5))
3   ;; eventually reaches ...
4   (add-complex (rectangular (real . 2) . 0)
5                (polar 3 . 5))
6   ;; which resulfs in ...
```

```
ice-9/boot-9.scm:1685:16: In procedure raise-exception:
In procedure +: Wrong type argument in position 1: (real . 2)
```

This issue was caused by not consuming the 'real tag when being coerced to a new type.

My code also had an issue where real numbers raised from rational couldn't be operated on because they were always nested inside the real number. This is solved by reducing the fraction to a decimal number, which fixes a few "bugs" in the book's code as well.

3.97 Exercise 2.85: `append-generic` with `drop`

3.97.1 Question

This section mentioned a method for "simplifying" a data object by lowering it in the tower of types as far as possible. Design a procedure `drop` that accomplishes this for the tower described in Exercise 2.83. The key is to decide, in some general way, whether an object can be lowered. For example, the complex number $1.5 + 0i$ can be lowered as far as `real`, the complex number $1 + 0i$ can be lowered as far as `integer`, and the complex number $2 + 3i$ cannot be lowered at all. Here is a plan for determining whether an object can be lowered: Begin by defining a generic operation `project` that "pushes" an object down in the tower. For example, projecting a complex number would involve throwing away the imaginary part. Then a number can be dropped if, when we `project` it and `raise` the result back to the type we started with, we end up with something equal to what we started with. Show how to implement this idea in detail, by writing a `drop` procedure that drops an object as far as possible. You will need to design the various projection operations[12] and install `project` as a generic operation in the system. You will also need to make use of a generic equality predicate, such as described in Exercise 2.79. Finally, use `drop` to rewrite `app⌋ly-generic` from Exercise 2.84 so that it "simplifies" its answers.

3.97.2 Answer

The question asks to install `project` as a generic operation in the system, and I assume that means a `project` function in the ops table. At first I thought it would be a function in the same way `raise` is, with coercions in the coercion table. However upon consideration, it might be for the purpose of handling the type tags, as I encountered in previous exercises.

```
(define (project x)
  (let ((t1 (type-tag x)))
    (let ((p (get 'project t1)))
      (if p
          (p (contents x))
          #f))))

(define (drop x)
  (if (not (pair? x))
```

---

[12]A real number can be projected to an integer using the `round` primitive, which returns the closest integer to its argument.

```
10           x
11         (let ((projected (project x)))
12           (cond ((eqv? projected #f)
13                  x)
14                 ((equ? x projected)
15                  (drop projected))
16                 (else x)))))

18  (put 'project 'rational
19        (lambda(num)
20          ((get 'numer '(rational))
21           num)))
22  (put 'project 'real
23        (lambda(num)
24          (round num)))
25  (put 'project 'complex
26        (lambda(num)
27          (make-real
28           ((get 'real-part '(complex)) num))))
```

```
1   (use-srfis '(1))
2   ;; Version of apply-generic that tries to drop its type
3   ;; when returning
4   (define (apply-generic op . args)
5     (define types (map type-tag args))
6     (define (map-raise highest-type item)
7       (if (equal? (type-tag item) highest-type)
8           item
9           (let ((raised (raise item)))
10            (if (not raised)
11                (error "MAP-RAISE: could not raise" item highest-type)
12                (map-raise highest-type raised)))))

14    (drop
15     (let ((proc (get op types)))
16       (if proc ; is this already doable?
17           (apply proc (map contents args))
18                                        ; if not, try type coercion

20           (let* ((highest-type (apply latest-in-tower types))
21                  (map-raise-prime (lambda(item)(map-raise highest-type
                    ↪  item)))
22                  (raised (map map-raise-prime args)))
23             (if (< (length raised) 3)
24                 ;; short args
25                 (let ((f (get op (map type-tag raised))))
26                   (if (eqv? #f f)
```

```
27                          (error
     ↳  "APPLY-GENERIC: could not find procedure for types"
     ↳    (map type-tag raised)
     ↳   raised)
28                          (apply f (map contents raised))))
29                  ;; long args
30                  (reduce (lambda(elem previous)
31                           (apply-generic op previous elem))
32                          "ERROR" raised)))))))
```

```
1    (load "mattcheck2.scm")
2    <<op-table>>
3    <<type-table>>
4    <<sqrt>>
5    <<square>>
6    <<echo>>
7    <<smarter-type-tagging>>
8    <<attach-tag-fixed>>
9    <<list-indices>>
10   <<tower-predicates>>
11   <<drop-project>>
12   <<apply-generic-drop>>
13   <<generic-arithmetic>>
14   <<scheme-number-package>>
15   <<real-package>>
16   <<equ?>>
17   <<eqzero?>>
18   <<real-package>>
19   <<basic-tower-coercions>>
20
21   (install-rational-package)
22   (install-rectangular-package)
23   (install-polar-package)
24   (install-complex-package)
25   (install-scheme-number-package)
26   (install-real-package)
27   (install-equ?)
28   (install-=zero?)
29   <<gen-math-tests>>
30
31   (mattcheck "raise success"
32             (raise 78)
33             (make-rational 78 1))
34   (mattcheck "raise fail"
35             (raise (make-complex-from-mag-ang 78 64))
36             #f)
37   (mattcheck "apply-generic 1"
38             (add 1 (make-rational 3 5))
```

```
39              (make-rational 8 5))
40   (mattcheck "apply-generic 2"
41              (add (make-real 2)
42                   (make-complex-from-mag-ang 3 5))
43              (make-complex-from-real-imag 2.850986556389679
             ↪  -2.8767728239894153))
44   (mattcheck "drop 1"
45              (drop (make-complex-from-real-imag 5 0))
46              5)
47   (mattcheck "drop 2"
48              (drop (make-real 5))
49              5)
50   (mattcheck "drop 3"
51              (drop (make-rational 5 1))
52              5)
```

```
5+5 = 10
5-5 = 0
5*5 = 25
5/5 = 1
1/2 + 1/2 = 1
1/2 - 1/2 = 0
1/2 * 1/2 = (rational 1 . 4)
1/2 / 1/2 = 1
3+4i + 3+4i = (complex rectangular 6 . 8)
3+4i - 3+4i = 0
3+4i * 3+4i = (complex polar 14.0 . 1.8545904360032244)
3+4i / 3+4i = 1.0
3m+4deg + 3m+4deg = (complex rectangular -3.921861725181672 . -4.540814971847569)
3m+4deg - 3m+4deg = 0.0
3m+4deg * 3m+4deg = (complex polar 9 . 8)
3m+4deg / 3m+4deg = 1
SUCCEED at raise success
SUCCEED at raise fail
SUCCEED at apply-generic 1
SUCCEED at apply-generic 2
SUCCEED at drop 1
SUCCEED at drop 2
SUCCEED at drop 3
```

Compare the gen-math-tests output here with that of the first run in Ex 2.77. They are being correctly simplified.

After getting these solutions working, I have to say I'm really reaping what I've sown. I've been very inconsistent with my tagging and untagging. Some functions in the type system expected tagged numbers, others expect untagged `contents`, which means those functions cannot be used in the same places. I've had to run over the system and make sure I'm doing it at the right time.

3.98 Exercise 2.86: complex numbers with flexible types

3.98.1 Question

Suppose we want to handle complex numbers whose real parts, imaginary parts, magnitudes, and angles can be either ordinary numbers, rational numbers, or other numbers we might wish to add to the system. Describe and implement the changes to the system needed to accommodate this. You will have to define operations such as `sine` and `cosine` that are generic over ordinary numbers and rational numbers.

3.98.2 Answer

We'll need the internal procedures in the complex package to themselves be calls to `apply-generic`, so that calculations with our new data types can be handled.

In the real and complex parts of the tower, I had assumed that the return of `real-part` would always be of tagged type `'real`. However, in this version it may be either real or rational, so I'll want to adjust the `project` procedures accordingly.

```
1   (define (add x y) (apply-generic 'add x y))
2   (define (sub x y) (apply-generic 'sub x y))
3   (define (mul x y) (apply-generic 'mul x y))
4   (define (div x y) (apply-generic 'div x y))
5   (define (real-part z) (apply-generic 'real-part z))
6   (define (imag-part z) (apply-generic 'imag-part z))
7   (define (magnitude z) (apply-generic 'magnitude z))
8   (define (angle z) (apply-generic 'angle z))
9   ;; Now we begin to change code for support
10  ;; These will be named slightly different so the primitive
11  ;; functions aren't shadowed
12  (define (square x) (mul x 2))
13  (define (sine x) (apply-generic 'sine x)) ;; (sin)
14  (define (cosine x) (apply-generic 'cosine x)); (cos)
15  (define (square-root x) (apply-generic 'square-root x));(sqrt)
16  (define (arctan x y) (apply-generic 'arctan x y));(atan)
17
18  (define (make-scheme-number n)
19    ((get 'make 'scheme-number) n))
20  (define (install-scheme-number-package)
21    (define (tag x) (attach-tag 'scheme-number x))
22    (put 'add '(scheme-number scheme-number)
23         (lambda (x y) (tag (+ x y))))
24    (put 'sub '(scheme-number scheme-number)
25         (lambda (x y) (tag (- x y))))
26    (put 'mul '(scheme-number scheme-number)
27         (lambda (x y) (tag (* x y))))
28    (put 'div '(scheme-number scheme-number)
```

```
29          (lambda (x y) (tag (/ x y))))
30     (put 'sine '(scheme-number)
31          (lambda (x) (tag (sin x))))
32     (put 'cosine '(scheme-number)
33          (lambda (x) (tag (cos x))))
34     (put 'square-root '(scheme-number)
35          (lambda (x) (tag (sqrt x))))
36     (put 'arctan '(scheme-number scheme-number)
37          (lambda (x y) (tag (atan x y))))
38     (put 'make 'scheme-number (lambda (x) (tag x)))
39     'done)
40
41  (define (install-rational-package)
42     ;; internal procedures
43     (define (numer x) (car x))
44     (define (denom x) (cdr x))
45     (define (make-rat n d)
46       (if (or (inexact? n)
47               (inexact? d))
48           (cons n d)
49           (let ((g (gcd n d)))
50             (cons (/ n g) (/ d g)))))
51     (define (add-rat x y)
52       (make-rat (+ (* (numer x) (denom y))
53                    (* (numer y) (denom x)))
54                 (* (denom x) (denom y))))
55     (define (sub-rat x y)
56       (make-rat (- (* (numer x) (denom y))
57                    (* (numer y) (denom x)))
58                 (* (denom x) (denom y))))
59     (define (mul-rat x y)
60       (make-rat (* (numer x) (numer y))
61                 (* (denom x) (denom y))))
62     (define (div-rat x y)
63       (make-rat (* (numer x) (denom y))
64                 (* (denom x) (numer y))))
65     ;; interface to rest of the system
66     (define (tag x) (attach-tag 'rational x))
67     (put 'add '(rational rational)
68          (lambda (x y) (tag (add-rat x y))))
69     (put 'sub '(rational rational)
70          (lambda (x y) (tag (sub-rat x y))))
71     (put 'mul '(rational rational)
72          (lambda (x y) (tag (mul-rat x y))))
73     (put 'div '(rational rational)
74          (lambda (x y) (tag (div-rat x y))))
75     (put 'make 'rational
76          (lambda (n d) (tag (make-rat n d))))
```

```scheme
77    (put 'numer '(rational) numer)
78    (put 'denom '(rational) denom)
79    (put 'sine '(rational)
80         (lambda(r) ((get 'sine '(scheme-number))
81                      (/ (numer r) (denom r)))))
82    (put 'cosine '(rational)
83         (lambda(r) ((get 'cosine '(scheme-number))
84                      (/ (numer r) (denom r)))))
85    (put 'square-root '(rational)
86         (lambda(r) ((get 'square-root '(scheme-number))
87                      (/ (numer r) (denom r)))))
88    (put 'arctan '(rational)
89         (lambda(r s) ((get 'arctan '(scheme-number))
90                        (/ (numer r) (denom r)) (/ (numer s) (denom s)))))
91    'done)
92
93  ;; for future use
94  (define (numer r)
95    ((get 'numer '(rational))
96     (contents r)))
97  (define (denom r)
98    ((get 'denom '(rational))
99     (contents r)))
100
101  (define (make-rational n d)
102    ((get 'make 'rational) n d))
103
104  (define (install-rectangular-package)
105    ;; internal procedures
106    (define (real-part z) (car z))
107    (define (imag-part z) (cdr z))
108    (define (make-from-real-imag x y) (cons x y))
109    (define (magnitude z)
110      (square-root (add (square (real-part z))
111                        (square (imag-part z)))))
112    (define (angle z)
113      (arctan (imag-part z) (real-part z)))
114    (define (make-from-mag-ang r a)
115      (cons (mul r (cosine a)) (mul r (sine a))))
116    ;; interface to the rest of the system
117    (define (tag x) (attach-tag 'rectangular x))
118    (put 'real-part '(rectangular) real-part)
119    (put 'imag-part '(rectangular) imag-part)
120    (put 'magnitude '(rectangular) magnitude)
121    (put 'angle '(rectangular) angle)
122    (put 'make-from-real-imag 'rectangular
123         (lambda (x y) (tag (make-from-real-imag x y))))
124    (put 'make-from-mag-ang 'rectangular
```

```scheme
125          (lambda (r a) (tag (make-from-mag-ang r a))))
126     'done)
127
128   (define (install-polar-package)
129     ;; internal procedures
130     (define (magnitude z) (car z))
131     (define (angle z) (cdr z))
132     (define (make-from-mag-ang r a) (cons r a))
133     (define (real-part z) (mul (magnitude z) (cosine (angle z))))
134     (define (imag-part z) (mul (magnitude z) (sine (angle z))))
135     (define (make-from-real-imag x y)
136       (cons (sqrt (+ (square x) (square y)))
137             (atan y x)))
138     ;; interface to the rest of the system
139     (define (tag x) (attach-tag 'polar x))
140     (put 'real-part '(polar) real-part)
141     (put 'imag-part '(polar) imag-part)
142     (put 'magnitude '(polar) magnitude)
143     (put 'angle '(polar) angle)
144     (put 'make-from-real-imag 'polar
145          (lambda (x y) (tag (make-from-real-imag x y))))
146     (put 'make-from-mag-ang 'polar
147          (lambda (r a) (tag (make-from-mag-ang r a))))
148     'done)
149
150   (define (install-complex-package)
151     (define (make-from-real-imag x y)
152       ((get 'make-from-real-imag 'rectangular) x y))
153     (define (make-from-mag-ang r a)
154       ((get 'make-from-mag-ang 'polar) r a))
155     ;; let's make these generic
156     (define (add-complex z1 z2)
157       (make-from-real-imag (add (real-part z1) (real-part z2))
158                            (add (imag-part z1) (imag-part z2))))
159     (define (sub-complex z1 z2)
160       (make-from-real-imag (sub (real-part z1) (real-part z2))
161                            (sub (imag-part z1) (imag-part z2))))
162     (define (mul-complex z1 z2)
163       (make-from-mag-ang (mul (magnitude z1) (magnitude z2))
164                          (add (angle z1) (angle z2))))
165     (define (div-complex z1 z2)
166       (make-from-mag-ang (div (magnitude z1) (magnitude z2))
167                          (sub (angle z1) (angle z2))))
168     ;; interface to rest of the system
169     (define (tag z) (attach-tag 'complex z))
170     (put 'add '(complex complex)
171          (lambda (z1 z2) (tag (add-complex z1 z2))))
172     (put 'sub '(complex complex)
```

```scheme
173            (lambda (z1 z2) (tag (sub-complex z1 z2)))))
174     (put 'mul '(complex complex)
175            (lambda (z1 z2) (tag (mul-complex z1 z2)))))
176     (put 'div '(complex complex)
177            (lambda (z1 z2) (tag (div-complex z1 z2)))))
178     (put 'make-from-real-imag 'complex
179            (lambda (x y) (tag (make-from-real-imag x y)))))
180     (put 'make-from-mag-ang 'complex
181            (lambda (r a) (tag (make-from-mag-ang r a)))))
182     (put 'real-part '(complex) real-part)
183     (put 'imag-part '(complex) imag-part)
184     (put 'magnitude '(complex) magnitude)
185     (put 'angle '(complex) angle))
186
187  (define (make-complex-from-real-imag x y)
188     ((get 'make-from-real-imag 'complex) x y))
189  (define (make-complex-from-mag-ang r a)
190     ((get 'make-from-mag-ang 'complex) r a))
191
192  (define (project x)
193     (let ((t1 (type-tag x)))
194        (let ((p (get 'project t1)))
195          (if p
196              (p (contents x))
197              #f))))
198
199  (define (drop x)
200     (if (not (pair? x))
201         x
202         (let ((projected (project x)))
203           (cond ((eqv? projected #f)
204                   x)
205                  ((equ? x projected)
206                   (drop projected))
207                  (else x)))))
208
209  (put 'project 'rational
210        (lambda(num)
211          ((get 'numer '(rational))
212           num)))
213  (put 'project 'real
214        (lambda(num)
215          (round num)))
216  ;; projecting a complex number may return something other than a single
     ↪ number
217  (put 'project 'complex
218        (lambda(num)
219          (let ((r ((get 'real-part '(complex)) num)))
```

```
220            (if (number? (contents r))
221                (make-real (contents r))
222                r))))
223
224   ;; equality testing with complex numbers also changes
225   (define (equ? x y) (apply-generic 'equ? x y))
226   (define (install-equ?)
227     (put 'equ? '(scheme-number scheme-number)
228          (lambda(x y) (= x y)))
229     (put 'equ? '(complex complex)
230          (lambda(x y)
231            (let ((t1 (type-tag x))
232                  (t2 (type-tag y)))
233              (cond ((and (eqv? t1 'polar)
234                          (eqv? t2 'polar))
235                     (and (equ? (apply-generic 'magnitude x)
236                                (apply-generic 'magnitude y))
237                          (equ? (apply-generic 'angle x)
238                                (apply-generic 'angle y))))
239                    (else
240                     (and (equ? (apply-generic 'real-part x)
241                                (apply-generic 'real-part y))
242                          (equ? (apply-generic 'imag-part x)
243                                (apply-generic 'imag-part y)))))))))
244     (put 'equ? '(rational rational)
245          (lambda(x y)
246            (and (= ((get 'numer '(rational)) x)
247                    ((get 'numer '(rational)) y))
248                 (= ((get 'denom '(rational)) x)
249                    ((get 'denom '(rational)) y)))))
250     (put 'equ? '(real real)
251          (lambda(x y) (= x y))))
```

```
1   (load "mattcheck2.scm")
2   <<op-table>>
3   <<type-table>>
4   <<sqrt>>
5   <<echo>>
6   <<smarter-type-tagging>>
7   <<attach-tag-fixed>>
8   <<list-indices>>
9   <<tower-predicates>>
10  <<apply-generic-drop>>
11  <<generic-arithmetic-complex-parts>>
12  <<real-package>>
13  <<eqzero?>>
14  <<real-package>>
15  <<basic-tower-coercions>>
```

293

```
16
17   (install-rational-package)
18   (install-rectangular-package)
19   (install-polar-package)
20   (install-complex-package)
21   (install-scheme-number-package)
22   (install-real-package)
23   (install-equ?)
24   (install-=zero?)
25   <<gen-math-tests>>
26
27   (mattcheck "raise success"
28             (raise 78)
29             (make-rational 78 1))
30   (mattcheck "raise fail"
31             (raise (make-complex-from-mag-ang 78 64))
32             #f)
33   (mattcheck "apply-generic 1"
34             (add 1 (make-rational 3 5))
35             (make-rational 8 5))
36   (mattcheck "apply-generic 2"
37             (add (make-real 2)
38                  (make-complex-from-mag-ang 3 5))
39             (make-complex-from-real-imag 2.850986556389679
             ↪  -2.8767728239894153))
40   (mattcheck "drop 1"
41             (drop (make-complex-from-real-imag 5 0))
42             5)
43   (mattcheck "drop 2"
44             (drop (make-real 5))
45             5)
46   (mattcheck "drop 3"
47             (drop (make-rational 5 1))
48             5)
49   (mattcheck "complex with nested rational"
50             (make-complex-from-real-imag 2 (make-rational 3 5))
51             (cons 'complex (cons 'rectangular (cons 2 (cons 'rational
             ↪  (cons 3 5))))))
52   (mattcheck "math on complex with nested rational"
53             (add (make-complex-from-real-imag 2 (make-rational 3 5))
54                  (make-complex-from-mag-ang (make-rational 3 5) 2))
55             (cons 'complex (cons 'rectangular
56                   (cons (cons 'rational (cons 8.751559490358574 5))
57                         (cons 'rational (cons 28.639461402385223 25))))))
```

```
(etc)
SUCCEED at complex with nested rational
SUCCEED at math on complex with nested rational
```

3.99   2.5: Symbolic Algebra

In this section, we implement an addition to the generic math system we have been building by adding support for polynomials (with only addition and multiplication involved) to start.

I'm going to split the polynomial package into two blocks so I can insert code inside without needing to copy the whole block again.

```
1   (define (install-polynomial-package)
2     ;; internal procedures
3     ;; representation of poly
4     (define (make-poly variable term-list) (cons variable term-list))
5     (define (variable p) (car p))
6     (define (terms p) (cdr p)) ;; renamed from term-list since many procs
        ↪  happen
7     (define (variable? x) (symbol? x))  ; to use "term-list" as an
        ↪  internal variable
8     (define (same-variable? v1 v2)
9       (and (variable? v1) (variable? v2) (eq? v1 v2)))
10    ;; representation of terms and term lists
11    (define (adjoin-term term term-list)
12      (if (=zero? (coeff term))
13          term-list
14          (cons term term-list)))
15    (define (the-empty-termlist) '())
16    (define (first-term term-list) (car term-list))
17    (define (rest-terms term-list) (cdr term-list))
18    (define (empty-termlist? term-list) (null? term-list))
19    (define (make-term order coeff) (list order coeff))
20    (define (order term) (car term))
21    (define (coeff term) (cadr term))
22    (define (add-poly p1 p2)
23      (if (same-variable? (variable p1) (variable p2))
24          (make-poly (variable p1)
25                     (add-terms (terms p1) (terms p2)))
26          (error "Polys not in same var: ADD-POLY" (list p1 p2))))
27    (define (mul-poly p1 p2)
28      (if (same-variable? (variable p1) (variable p2))
29          (make-poly (variable p1)
30                     (mul-terms (terms p1) (terms p2)))
31          (error "Polys not in same var: MUL-POLY" (list p1 p2))))
32    (define (add-terms L1 L2)
33      (cond ((empty-termlist? L1) L2)
34            ((empty-termlist? L2) L1)
35            (else
36             (let ((t1 (first-term L1))
37                   (t2 (first-term L2)))
38               (cond ((> (order t1) (order t2))
39                      (adjoin-term
```

```
40                        t1 (add-terms (rest-terms L1) L2)))
41                     ((< (order t1) (order t2))
42                      (adjoin-term
43                       t2 (add-terms L1 (rest-terms L2))))
44                     (else
45                      (adjoin-term
46                       (make-term (order t1)
47                                  (add (coeff t1) (coeff t2)))
48                       (add-terms (rest-terms L1)
49                                  (rest-terms L2)))))))))))
50   (define (mul-terms L1 L2)
51     (if (empty-termlist? L1)
52         (the-empty-termlist)
53         (add-terms (mul-term-by-all-terms (first-term L1) L2)
54                    (mul-terms (rest-terms L1) L2))))
55   ;; interface to rest of the system
56   (define (tag p) (attach-tag 'polynomial p))
57   (put 'add '(polynomial polynomial)
58        (lambda (p1 p2) (tag (add-poly p1 p2))))
59   (put 'mul '(polynomial polynomial)
60        (lambda (p1 p2) (tag (mul-poly p1 p2))))
61   (put 'make 'polynomial
62        (lambda (var terms) (tag (make-poly var terms))))
63
```

```
1   (define (mul-term-by-all-terms t1 L)
2     (if (empty-termlist? L)
3         (the-empty-termlist)
4         (let ((t2 (first-term L)))
5           (adjoin-term
6            (make-term (+ (order t1) (order t2))
7                       (mul (coeff t1) (coeff t2)))
8            (mul-term-by-all-terms t1 (rest-terms L))))))
```

```
1    'done)
2
3  (define (make-polynomial var terms)
4    ((get 'make 'polynomial) var terms))
```

3.100   Exercise 2.87: `=zero?` for polynomials

3.100.1   Question

Install `=zero?` for polynomials in the generic arithmetic package. This will
allow `adjoin-term` to work for polynomials with coefficients that are themselves
polynomials.

296

3.100.2 Answer

After testing, I see a problem. `add`ing nested polynomials works fine. However, when a coefficient is a polynomial, `mul-term-by-all-terms` will call `mul` with the types `'(polynomial scheme-number)`. If I added polynomials to the type tower and used `raise` to make it into a `'(polynomial polynomial)` call, I then need to specify a matching variable for the raised number, which is not available in `raise`. It seems like I'll need to modify the `mul-term-by-all-terms` procedure to detect when one item is a polynomial and the other isn't, and turn the non-polynomial entry into a polynomial before calling `mul`.

```scheme
(define (if-1st-poly-make-2nd-poly t1 t2)
  (if (and (eqv? 'polynomial (type-tag t1))
           (not (eqv? 'polynomial (type-tag t2))))
      (tag (make-poly (variable (contents t1)) (list (make-term 0 t2))))
      t2))
(define (mul-term-by-all-terms t1 L)
  (if (empty-termlist? L)
      (the-empty-termlist)
      (let* ((t2 (first-term L)) ;; If one coeff is a poly, make the
             ↪  other a poly
             (t1-coeff (if-1st-poly-make-2nd-poly (coeff t2) (coeff t1)))
             (t2-coeff (if-1st-poly-make-2nd-poly (coeff t1) (coeff
                        ↪  t2))))

        (adjoin-term
         (make-term (+ (order t1) (order t2))
                    (mul t1-coeff t2-coeff))
         (mul-term-by-all-terms t1 (rest-terms L))))))
```

```scheme
(define (terms-=zero? L)
  (or (empty-termlist? L)
      (and (=zero? (coeff (first-term L)))
           (terms-=zero? (rest-terms L)))))
(define (poly-=zero? p)
  (terms-=zero? (terms p)))
(put '=zero? '(polynomial) poly-=zero?)
```

```scheme
(load "mattcheck2.scm")
<<op-table>>
<<type-table>>
<<sqrt>>
<<echo>>
<<smarter-type-tagging>>
<<attach-tag-fixed>>
<<list-indices>>
<<tower-predicates>>
```

297

```
10  <<apply-generic-drop>>
11  <<generic-arithmetic-complex-parts>>
12  <<real-package>>
13  <<eqzero?>>
14  <<real-package>>
15  <<basic-tower-coercions>>
16
17  <<polynomials-package-top>>
18  <<mul-term-by-all-terms-mine>>
19  <<poly-terms-eqzero>>
20  <<polynomials-package-bottom>>
21
22  (install-rational-package)
23  (install-rectangular-package)
24  (install-polar-package)
25  (install-complex-package)
26  (install-scheme-number-package)
27  (install-real-package)
28  (install-equ?)
29  (install-=zero?)
30  (install-polynomial-package)
31  <<gen-math-tests>>
32
33  (mattcheck "raise success"
34            (raise 78)
35            (make-rational 78 1))
36  (mattcheck "raise fail"
37            (raise (make-complex-from-mag-ang 78 64))
38            #f)
39  (mattcheck "apply-generic 1"
40            (add 1 (make-rational 3 5))
41            (make-rational 8 5))
42  (mattcheck "apply-generic 2"
43            (add (make-real 2)
44                 (make-complex-from-mag-ang 3 5))
45            (make-complex-from-real-imag 2.850986556389679
                 ↪  -2.8767728239894153))
46  (mattcheck "drop 1"
47            (drop (make-complex-from-real-imag 5 0))
48            5)
49  (mattcheck "drop 2"
50            (drop (make-real 5))
51            5)
52  (mattcheck "drop 3"
53            (drop (make-rational 5 1))
54            5)
55  (mattcheck "complex with nested rational"
56            (make-complex-from-real-imag 2 (make-rational 3 5))
```

298

```
57              (cons 'complex (cons 'rectangular (cons 2 (cons 'rational
    ↪  (cons 3 5))))))
58  (mattcheck "math on complex with nested rational"
59              (add (make-complex-from-real-imag 2 (make-rational 3 5))
60                    (make-complex-from-mag-ang (make-rational 3 5) 2))
61              (cons 'complex (cons 'rectangular
62                                      (cons (cons 'rational (cons
    ↪  8.751559490358574 5))
63                                          (cons 'rational (cons
    ↪  28.639461402385223 25))))))
64  (mattcheck "polynomial = zero"
65              (=zero? (make-polynomial 'x '((0 0) (0 0)))))
66  (mattcheck "polynomial ≠ zero" #f
67              (=zero? (make-polynomial 'x '((0 0) (1 1)))))
68  (let ((p1 '((100 1)(2 2)(0 1)))
69        (p2 (list (list 7 (make-polynomial 'x '((3 2)))))))
70    (mattcheck "polynomial addition"
71              (add (make-polynomial 'x p1)
72                    (make-polynomial 'x p1))
73              '(polynomial x (100 2)(2 4)(0 2)))
74    (mattcheck "polynomial multiplication"
75              (mul (make-polynomial 'x p1)
76                    (make-polynomial 'x p1))
77              '(polynomial x (200 1) (102 4) (100 2) (4 4) (2 4) (0 1)))
78    (mattcheck "nested polynomial addition"
79              (add (make-polynomial 'x p2)
80                    (make-polynomial 'x p1))
81              '(polynomial x (100 1) (7 (polynomial x (3 2))) (2 2) (0
    ↪  1)))
82    (mattcheck "nested polynomial mult"
83              (mul (make-polynomial 'x p2)
84                    (make-polynomial 'x p1))
85              '(polynomial x (107 (polynomial x (3 2))) (9 (polynomial x
    ↪  (3 4))) (7 (polynomial x (3 2))))))
```

```
(etc ... )
SUCCEED at polynomial = zero
SUCCEED at polynomial ≠ zero
SUCCEED at polynomial addition
SUCCEED at polynomial multiplication
SUCCEED at nested polynomial addition
SUCCEED at nested polynomial mult
```

3.101   Exercise 2.88: negation, subtracting polynomials

3.101.1   Question

Extend the polynomial system to include subtraction of polynomials. (Hint:
You may find it helpful to define a generic negation operation.)

3.101.2 Answer

```
1  (define (neg x)
2    (apply-generic 'neg x))
3  (define (install-negation)
4    (put 'neg '(scheme-number)
5         (lambda(x) (- x)))
6    (put 'neg '(rational)
7         (lambda(R) (make-rational (neg ((get 'numer '(rational)) R))
8                                   ((get 'denom '(rational)) R))))
9    (put 'neg '(real) (get 'neg '(scheme-number)))
10   (put 'neg '(rectangular)
11        (lambda (C)
12          (make-complex-from-real-imag (neg ((get 'real-part
               ↪ '(rectangular)) C))
13                                       (neg ((get 'imag-part
                                          ↪ '(rectangular)) C)))))
14   (put 'neg '(polar)
15        (lambda (C)
16          (make-complex-from-mag-ang (neg ((get 'magnitude '(polar)) C))
17                                     ((get 'angle '(polar)) C))))
18   (put 'neg '(complex)
19        (lambda(C) (apply-generic 'neg C))))
```

```
1      ;; inserted in install routine
2      (define (neg-poly p)
3      (make-poly (variable p)
4                 (map (lambda(t)
5                        (make-term (order t) (neg (coeff t))))
6                      (terms p))))
7      (put 'neg '(polynomial)
8           (lambda (p1) (tag (neg-poly p1))))
9      (define (sub-poly p1 p2)
10     (if (same-variable? (variable p1) (variable p2))
11         (add-poly p1 (neg-poly p2))
12         (error "Polys not in same var: SUB-POLY" (list p1 p2))))
13     (put 'sub '(polynomial polynomial)
14          (lambda (p1 p2) (tag (sub-poly p1 p2))))
```

```
1  (load "mattcheck2.scm")
2  <<op-table>>
3  <<type-table>>
4  <<sqrt>>
5  <<echo>>
6  <<smarter-type-tagging>>
7  <<attach-tag-fixed>>
8  <<list-indices>>
```

```
 9  <<tower-predicates>>
10  <<apply-generic-drop>>
11  <<generic-arithmetic-complex-parts>>
12  <<real-package>>
13  <<eqzero?>>
14  <<real-package>>
15  <<basic-tower-coercions>>
16
17  <<polynomials-package-top>>
18  <<mul-term-by-all-terms-mine>>
19  <<poly-terms-eqzero>>
20  <<polynomial-subtraction>>
21  <<polynomials-package-bottom>>
22
23  <<generic-negation>>
24
25  (install-rational-package)
26  (install-rectangular-package)
27  (install-polar-package)
28  (install-complex-package)
29  (install-scheme-number-package)
30  (install-real-package)
31  (install-equ?)
32  (install-=zero?)
33  (install-polynomial-package)
34  (install-negation)
35
36  <<gen-math-tests>>
37
38  (mattcheck "raise success"
39            (raise 78)
40            (make-rational 78 1))
41  (mattcheck "raise fail"
42            (raise (make-complex-from-mag-ang 78 64))
43            #f)
44  (mattcheck "apply-generic 1"
45            (add 1 (make-rational 3 5))
46            (make-rational 8 5))
47  (mattcheck "apply-generic 2"
48            (add (make-real 2)
49                 (make-complex-from-mag-ang 3 5))
50            (make-complex-from-real-imag 2.850986556389679
              ↪ -2.8767728239894153))
51  (mattcheck "drop 1"
52            (drop (make-complex-from-real-imag 5 0))
53            5)
54  (mattcheck "drop 2"
55            (drop (make-real 5))
```

```
56              5)
57  (mattcheck "drop 3"
58              (drop (make-rational 5 1))
59              5)
60  (mattcheck "complex with nested rational"
61              (make-complex-from-real-imag 2 (make-rational 3 5))
62              (cons 'complex (cons 'rectangular (cons 2 (cons 'rational
        ↪  (cons 3 5))))))
63  (mattcheck "math on complex with nested rational"
64              (add (make-complex-from-real-imag 2 (make-rational 3 5))
65                   (make-complex-from-mag-ang (make-rational 3 5) 2))
66              (cons 'complex (cons 'rectangular
67                                  (cons (cons 'rational (cons
        ↪  8.751559490358574 5))
68                                       (cons 'rational (cons
        ↪  28.639461402385223 25))))))
69  (mattcheck "polynomial = zero"
70              (=zero? (make-polynomial 'x '((0 0) (0 0)))))
71  (mattcheck "polynomial ≠ zero" #f
72              (=zero? (make-polynomial 'x '((0 0) (1 1)))))
73  (let ((p1 '((100 1)(2 2)(0 1)))
74        (p2 (list (list 7 (make-polynomial 'x '((3 2)))))))
75    (mattcheck "polynomial addition"
76                (add (make-polynomial 'x p1)
77                     (make-polynomial 'x p1))
78                '(polynomial x (100 2)(2 4)(0 2)))
79    (mattcheck "polynomial multiplication"
80                (mul (make-polynomial 'x p1)
81                     (make-polynomial 'x p1))
82                '(polynomial x (200 1) (102 4) (100 2) (4 4) (2 4) (0 1)))
83    (mattcheck "nested polynomial addition"
84                (add (make-polynomial 'x p2)
85                     (make-polynomial 'x p1))
86                '(polynomial x (100 1) (7 (polynomial x (3 2))) (2 2) (0
        ↪  1)))
87    (mattcheck "nested polynomial mult"
88                (mul (make-polynomial 'x p2)
89                     (make-polynomial 'x p1))
90                '(polynomial x (107 (polynomial x (3 2))) (9 (polynomial x
        ↪  (3 4))) (7 (polynomial x (3 2))))))
91  (mattcheck "negation scheme-number"
92              (neg 5) -5)
93  (mattcheck "negation rational"
94              (neg (make-rational 1 5))
95              (make-rational -1 5))
96  (mattcheck "negation rectangular complex"
97              (neg (make-complex-from-real-imag 3 7))
98              (make-complex-from-real-imag -3 -7))
```

```
 99   (mattcheck "negation polar complex"
100              (neg (make-complex-from-mag-ang 3 7))
101              (make-complex-from-mag-ang -3 7))
102
103   (let ((p1 '((100 1)(2 2)(0 1)))
104         (p2 (list (list 7 (make-polynomial 'x '((3 2)))))))
105         (p3 '((100 1)(2 1)(0 3)))
106         (p4 '((100 -1)(2 -2)(0 -1))))
107     (mattcheck "negation polynomials"
108                (neg (make-polynomial 'x p1))
109                (make-polynomial 'x p4))
110     (mattcheck "polynomial subtraction"
111                (sub (make-polynomial 'x p1)
112                     (make-polynomial 'x p3))
113                '(polynomial x (2 1) (0 -2))))
```

```
(etc … )
SUCCEED at negation scheme-number
SUCCEED at negation rational
SUCCEED at negation rectangular complex
SUCCEED at negation polar complex
SUCCEED at negation polynomials
SUCCEED at polynomial subtraction
```

3.102   Exercise 2.89: an abstraction for dense polynomials

3.102.1   Question

Define procedures that implement the term-list representation described above
as appropriate for dense polynomials.

3.102.2   Answer

At first I thought I needed to rewrite the `order` and `coeff` procedures. Then
I realized these are only called with one term, and thus don't have the term
list available, which means the order can't be learned. It's much simpler to
make a new `first-term` selector that gets the needed info and always returns
the sparse term format, so long as `adjoin-set` is modified so it will convert back
when adding to a sparse term list.

```
1   (define (sparse-termlist? term-list)
2     (let ((t (car term-list)))
3       (cond ((and (list? t)
4                   (number? (first t))
5                   (number? (second t))) #t)
6             ((number? t) #f)
7             (else (error "badly formed term: SPARSE-TERMLIST?" t
                ↪ term-list)))))
```

```
 8  (define (make-sparse-term order coeff)
 9    (list order coeff))
10  (define (dense-first-term term-list)
11    (make-term (car term-list)
12               (1- (length term-list))))
13  (define (sparse-first-term term-list)
14    (car term-list))
15  (define (first-term term-list)
16    (if (sparse-termlist? term-list)
17        (sparse-first-term term-list)
18        (dense-first-term term-list)))
```

3.103 Exercise 2.90: dense polynomials integrated into the system

3.103.1 Question

Suppose we want to have a polynomial system that is efficient for both sparse and dense polynomials. One way to do this is to allow both kinds of term-list representations in our system. The situation is analogous to the complex-number example of 2.4: Multiple Representations for Abstract Data, where we allowed both rectangular and polar representations. To do this we must distinguish different types of term lists and make the operations on term lists generic. Redesign the polynomial system to implement this generalization. This is a major effort, not a local change.

3.103.2 Answer

Although I was debating using tags to distinguish the two types, it occurs to me they can be told apart just by checking the contents. A valid coefficient is either a number or a list for a nested type. If the first item of the list is a list whose `car` is a number and whose `cadr` is a number or list, assume it's sparse. On the other hand, if the first item is a number, or list that doesn't match that previous description, assume it's dense. This format would *not* be adequate if we had any potential nested types that happened to look similar, and especially not if the term-lists were meant for external use.

The new implementations are abstracted such that average procedures using `first-term`, `rest-term` and `adjoin-term` are totally unaffected.

Whenever a procedure requests a `first-term`, a sparse representation is returned (pair of (`order` `coeff`)). Then, when `adjoin-term` is run, the list is appended-to appropriately whether it is sparse or dense. When `make-poly` is run, the statistics of the term list are measured to decide which implementation should be used. If there are more zero-terms than $\frac{1}{2}$ of the highest order in the term, the sparse implementation is used.

```
 1  (use-srfis '(1))
```

304

```scheme
(define (install-polynomial-package)
  ;; internal procedures
  ;; representation of poly
  (define (make-poly variable term-list)
    (cons variable
          (if (empty-termlist? term-list)
              term-list
              (cond ((and (sparse-termlist? term-list)
                          (not (should-be-sparse? term-list)))
                     (sparse-to-dense term-list))
                    ((and (not (sparse-termlist? term-list))
                          (should-be-sparse? term-list))
                     (dense-to-sparse term-list))
                    (else term-list)))))
  (define (variable p) (car p))
  (define (terms p) (cdr p))
  (define (variable? x) (symbol? x))
  (define (same-variable? v1 v2)
    (and (variable? v1) (variable? v2) (eq? v1 v2)))

  ;; representation of terms and term lists
  (define (sparse-termlist? term-list)
    (let ((t (car term-list)))
      (cond ((and (list? t) ;; looks like a sparse list
                  (number? (first t))
                  (or (list? (second t)) ;; or sparse with nested poly
                      (number? (second t)))) #t)
            ((or (number? t) ;; looks dense
                 (list? t)) #f) ;; or dense with a nested poly
            (else (error "badly formed term: SPARSE-TERMLIST?" t
              ↪ term-list)))))
  (define (should-be-sparse? term-list)
    ;; sparse or dense term-list -> bool
    ;; if more than half of term-list is zeroes, return true
    ;; reasoning:
    ;; - number of elements used for sparse list is (non-zero elements *
      ↪  2)
    ;; - number of elements used for dense list is (highest order)
    (define (count-zeroes term-list)
      ;; term-list -> (highest order, zeroes)
      (let iter ((length 0)
                 (zeroes 0)
                 (TL (if (sparse-termlist? term-list)
                         (sparse-to-dense term-list)
                         term-list)))
        (if (empty-termlist? TL)
            (list zeroes length)
            (iter (1+ length)
```

```scheme
48                      (if (=zero? (car TL))
49                          (1+ zeroes)
50                          zeroes)
51                      (cdr TL)))))
52      (let ((stats (if (sparse-termlist? term-list) ;; do appropriate
    ↪  measurement
53                       (list (- (1+ (order (sparse-first-term term-list)))
54                                (length term-list))
55                             (1+ (order (sparse-first-term term-list))))
56                       (count-zeroes term-list))))
57        (let ((zero-elements (first stats))
58              (highest-order (second stats)))
59          (> (/ zero-elements highest-order)
60             (/ 1 2)))))
61    (define (sparse-to-dense term-list)
62      (let iter ((old-list (reverse term-list))
63                 (new-list (the-empty-termlist)))
64        (if (empty-termlist? old-list)
65            new-list
66            (iter (rest-terms old-list)
67                  (adjoin-term-dense (sparse-first-term old-list)
68                                     new-list)))))
69    (define (dense-to-sparse term-list)
70      (let iter ((old-list (reverse term-list))
71                 (old-order 0)
72                 (new-list (the-empty-termlist)))
73        (if (empty-termlist? old-list)
74            new-list
75            (iter (rest-terms old-list)
76                  (1+ old-order)
77                  (adjoin-term-sparse (make-sparse-term old-order
78                                                        (coeff (dense-firs⌋
                                                        ↪  t-term
                                                        ↪  old-list)))
79                                      new-list)))))
80    (define (make-sparse-term order coeff)
81      (list order coeff))
82    (define (dense-termlist-order term-list)
83      (1- (length term-list)))
84    (define (dense-first-term term-list)
85      (make-sparse-term (dense-termlist-order term-list)
86                        (car term-list)))
87    (define (sparse-first-term term-list)
88      (car term-list))
89    (define (first-term term-list)
90      (if (sparse-termlist? term-list)
91          (sparse-first-term term-list)
92          (dense-first-term term-list)))
```

```scheme
93    (define (adjoin-term-sparse term term-list)
94      (cons term term-list))
95    (define (adjoin-term-dense term term-list)
96      ;; if term is exactly 1 order above top of list, adjoin
97      ;; if term is more than 1 order above, adjoin 0 and recurse
98      (let ((top-order (dense-termlist-order term-list)))
99        (cond ((= (1+ top-order) (order term))
100               (cons (coeff term) term-list))
101              ((≤ top-order (order term))
102               (adjoin-term-dense term (cons 0 term-list)))
103              (else (error
                 ↪ "term is lower order than top of list: ADJOIN-TERM-DENSE"
                 ↪   term term-list
                 ↪  top-order)))))

104
105   ;;  ;; I wrote the versions below which can insert into the middle of
      ↪  the list,
106   ;;  ;; before remembering that a footnote in the text excuses this.
107   ;;
108   ;; (define (adjoin-term-sparse term term-list)
109   ;;    ;; if term is higher order, cons to list
110   ;;    ;; if term is lower order than top of list, try to insert in the
      ↪  middle
111   ;;    (let ((top-order (order (sparse-first-term term-list))))
112   ;;      (cond ((≤ (1+ top-order) (order term))
113   ;;              (cons term term-list))
114   ;;            (else (adjoin-term-sparse (sparse-first-term term-list)
115   ;;                                      (adjoin-term-sparse term
116   ;;                                                          (rest-terms
      ↪  term-list)))))))
117   ;; (define (adjoin-term-dense term term-list)
118   ;;    ;; if term is 1 order above top of list, adjoin
119   ;;    ;; if term is more than 1 order above, adjoin 0 and recurse
120   ;;    ;; if term is lower order than top of list, try to insert in the
      ↪  middle
121   ;;    (let ((top-order (dense-termlist-order term-list)))
122   ;;      (cond ((= (1+ top-order) (order term))
123   ;;              (cons (coeff term) term-list))
124   ;;            ((≤ top-order (order term))
125   ;;             (adjoin-term-dense term (cons 0 term-list)))
126   ;;            (else (adjoin-term-dense (dense-first-term term-list)
127   ;;                                     (adjoin-term-dense term
128   ;;                                                        (rest-terms
      ↪  term-list)))))))

129
130   (define (adjoin-term term term-list)
131     (if (=zero? (coeff term))
132         term-list
```

```scheme
133              (if (or (empty-termlist? term-list)
134                      (sparse-termlist? term-list))
135                  (adjoin-term-sparse term term-list)
136                  (adjoin-term-dense term term-list)))))
137     (define (the-empty-termlist) '())
138     (define (rest-terms term-list) (cdr term-list))
139     (define (empty-termlist? term-list) (null? term-list))
140     (define (order term) (first term))
141     (define (coeff term) (second term))
142     (define (add-poly p1 p2)
143       (if (same-variable? (variable p1) (variable p2))
144           (make-poly (variable p1)
145                      (add-terms (terms p1) (terms p2)))
146           (error "Polys not in same var: ADD-POLY" (list p1 p2))))
147     (define (mul-poly p1 p2)
148       (if (same-variable? (variable p1) (variable p2))
149           (make-poly (variable p1)
150                      (mul-terms (terms p1) (terms p2)))
151           (error "Polys not in same var: MUL-POLY" (list p1 p2))))
152     (define (add-terms L1 L2)
153       (cond ((empty-termlist? L1) L2)
154             ((empty-termlist? L2) L1)
155             (else
156              (let ((t1 (first-term L1))
157                    (t2 (first-term L2)))
158                (cond ((> (order t1) (order t2))
159                       (adjoin-term
160                        t1 (add-terms (rest-terms L1) L2)))
161                      ((< (order t1) (order t2))
162                       (adjoin-term
163                        t2 (add-terms L1 (rest-terms L2))))
164                      (else
165                       (adjoin-term
166                        (make-sparse-term (order t1)
167                                          (add (coeff t1) (coeff t2)))
168                        (add-terms (rest-terms L1)
169                                   (rest-terms L2)))))))))
170     (define (mul-terms L1 L2)
171       (if (empty-termlist? L1)
172           (the-empty-termlist)
173           (add-terms (mul-term-by-all-terms (first-term L1) L2)
174                      (mul-terms (rest-terms L1) L2))))
175
176     ;; interface to rest of the system
177     (define (tag p) (attach-tag 'polynomial p))
178     (put 'add '(polynomial polynomial)
179          (lambda (p1 p2) (tag (add-poly p1 p2))))
180     (put 'mul '(polynomial polynomial)
```

```scheme
181         (lambda (p1 p2) (tag (mul-poly p1 p2)))))
182    (put 'make 'polynomial
183         (lambda (var terms) (tag (make-poly var terms))))
184    (define (if-1st-poly-make-2nd-poly t1 t2)
185      (if (and (eqv? 'polynomial (type-tag t1))
186               (not (eqv? 'polynomial (type-tag t2))))
187          (tag (make-poly (variable (contents t1)) (list t2)))
188          t2))
189    (define (mul-term-by-all-terms t1 L)
190      (if (empty-termlist? L)
191          (the-empty-termlist)
192          (let* ((t2 (first-term L)) ;; If one coeff is a poly, make the
           ↪  other a poly
193                 (t1-coeff (if-1st-poly-make-2nd-poly (coeff t2) (coeff
                   ↪  t1)))
194                 (t2-coeff (if-1st-poly-make-2nd-poly (coeff t1) (coeff
                   ↪  t2))))

196            (adjoin-term
197             (make-sparse-term (+ (order t1) (order t2))
198                               (mul t1-coeff t2-coeff))
199             (mul-term-by-all-terms t1 (rest-terms L))))))
200    (define (terms-=zero? L)
201      (or (empty-termlist? L)
202          (and (=zero? (coeff (first-term L)))
203               (terms-=zero? (rest-terms L)))))
204    (define (poly-=zero? p)
205      (terms-=zero? (terms p)))
206    (put '=zero? '(polynomial) poly-=zero?)
207    (define (neg-terms TL)
208      (let ((negterm (if (sparse-termlist? TL)
209                         (lambda(t)
210                           (make-sparse-term (order t) (neg (coeff t))))
211                         (lambda(t)
212                           (neg (coeff t))))))
213        (map negterm TL)))
214    (define (sub-terms T1 T2)
215      (add-terms T1 (neg-terms T2)))
216    (define (neg-poly p)
217      (make-poly (variable p)
218                 (neg-terms (terms p))))
219    (put 'neg '(polynomial)
220         (lambda (p1) (tag (neg-poly p1))))
221    (define (sub-poly p1 p2)
222      (add-poly p1 (neg-poly p2)))
223    (put 'sub '(polynomial polynomial)
224         (lambda (p1 p2) (tag (sub-poly p1 p2))))
```

309

```
1      'done)
2   (define (make-polynomial var terms)
3     ((get 'make 'polynomial) var terms))
```

```
1   (load "mattcheck2.scm")
2   <<op-table>>
3   <<type-table>>
4   <<sqrt>>
5   <<echo>>
6   <<smarter-type-tagging>>
7   <<attach-tag-fixed>>
8   <<list-indices>>
9   <<tower-predicates>>
10  <<apply-generic-drop>>
11  <<generic-arithmetic-complex-parts>>
12  <<real-package>>
13  <<eqzero?>>
14  <<real-package>>
15  <<basic-tower-coercions>>
16
17  <<polynomial-sparse-dense-top>>
18  <<polynomial-sparse-dense-bottom>>
19
20  <<generic-negation>>
21
22  (install-rational-package)
23  (install-rectangular-package)
24  (install-polar-package)
25  (install-complex-package)
26  (install-scheme-number-package)
27  (install-real-package)
28  (install-equ?)
29  (install-=zero?)
30  (install-polynomial-package)
31  (install-negation)
32
33  <<gen-math-tests>>
34
35  (mattcheck "raise success"
36             (raise 78)
37             (make-rational 78 1))
38  (mattcheck "raise fail"
39             (raise (make-complex-from-mag-ang 78 64))
40             #f)
41  (mattcheck "apply-generic 1"
42             (add 1 (make-rational 3 5))
43             (make-rational 8 5))
```

```
44  (mattcheck "apply-generic 2"
45              (add (make-real 2)
46                   (make-complex-from-mag-ang 3 5))
47              (make-complex-from-real-imag 2.850986556389679
               ↪   -2.8767728239894153))
48  (mattcheck "drop 1"
49              (drop (make-complex-from-real-imag 5 0))
50              5)
51  (mattcheck "drop 2"
52              (drop (make-real 5))
53              5)
54  (mattcheck "drop 3"
55              (drop (make-rational 5 1))
56              5)
57  (mattcheck "complex with nested rational"
58              (make-complex-from-real-imag 2 (make-rational 3 5))
59              (cons 'complex (cons 'rectangular (cons 2 (cons 'rational
               ↪   (cons 3 5))))))
60  (mattcheck "math on complex with nested rational"
61              (add (make-complex-from-real-imag 2 (make-rational 3 5))
62                   (make-complex-from-mag-ang (make-rational 3 5) 2))
63              (cons 'complex (cons 'rectangular
64                                  (cons (cons 'rational (cons
                                     ↪   8.751559490358574 5))
65                                        (cons 'rational (cons
                                     ↪   28.639461402385223 25))))))
66  (mattcheck "polynomial = zero"
67              (=zero? (make-polynomial 'x '((0 0)))))
68  (mattcheck "polynomial ≠ zero" #f
69              (=zero? (make-polynomial 'x '((1 1) (0 1)))))
70
71  (let ((p1 '((100 1)(2 2)(0 1)))
72        (p2 (list (list 7 (make-polynomial 'x '((3 2))))))
73        (p3 '((100 1)(2 1)(0 3)))
74        (p4 '((100 -1)(2 -2)(0 -1)))
75        (p5 '(8 2 5 -3 -7)))
76    (mattcheck "polynomial addition"
77              (add (make-polynomial 'x p1)
78                   (make-polynomial 'x p1))
79              '(polynomial x (100 2)(2 4)(0 2)))
80    (mattcheck "polynomial multiplication"
81              (mul (make-polynomial 'x p1)
82                   (make-polynomial 'x p1))
83              '(polynomial x (200 1) (102 4) (100 2) (4 4) (2 4) (0 1)))
84    (mattcheck "nested polynomial addition"
85              (add (make-polynomial 'x p2)
86                   (make-polynomial 'x p1))
```

311

```
87            '(polynomial x (100 1) (7 (polynomial x (3 2))) (2 2) (0
              ↪  1)))
88    (mattcheck "nested polynomial mult"
89            (mul (make-polynomial 'x p2)
90                 (make-polynomial 'x p1))
91            '(polynomial x (107 (polynomial x (3 2))) (9 (polynomial x
              ↪  (3 4))) (7 (polynomial x (3 2)))))
92    (mattcheck "negation scheme-number"
93            (neg 5) -5)
94    (mattcheck "negation rational"
95            (neg (make-rational 1 5))
96            (make-rational -1 5))
97    (mattcheck "negation rectangular complex"
98            (neg (make-complex-from-real-imag 3 7))
99            (make-complex-from-real-imag -3 -7))
100   (mattcheck "negation polar complex"
101            (neg (make-complex-from-mag-ang 3 7))
102            (make-complex-from-mag-ang -3 7))
103
104   (mattcheck "negation polynomials"
105            (neg (make-polynomial 'x p1))
106            (make-polynomial 'x p4))
107   (mattcheck "polynomial subtraction"
108            (sub (make-polynomial 'x p1)
109                 (make-polynomial 'x p3))
110            '(polynomial x 1 0 -2)))
```

```
(etc ... )
SUCCEED at negation polynomials
SUCCEED at polynomial subtraction
```

3.104 Exercise 2.91: division of polynomials

3.104.1 Question

A univariate polynomial can be divided by another one to produce a polynomial quotient and a polynomial remainder. For example,

$\frac{x^5-1}{x^2-1} = x^3 + x$, remainder $x - 1$

Division can be performed via long division. That is, divide the highest-order term of the dividend by the highest-order term of the divisor. The result is the first term of the quotient. Next, multiply the result by the divisor, subtract that from the dividend, and produce the rest of the answer by recursively dividing the difference by the divisor. Stop when the order of the divisor exceeds the order of the dividend and declare the dividend to be the remainder. Also, if the dividend ever becomes zero, return zero as both quotient and remainder.

We can design a `div-poly` procedure on the model of `add-poly` and `mul-⌋ poly`. The procedure checks to see if the two polys have the same variable. If so, `div-poly` strips off the variable and passes the problem to `div-terms`, which

performs the division operation on term lists. `div-poly` finally reattaches the variable to the result supplied by `div-terms`. It is convenient to design `div-t⌋ erms` to compute both the quotient and the remainder of a division. `div-terms` can take two term lists as arguments and return a list of the quotient term list and the remainder term list.

Complete the following definition of `div-terms` by filling in the missing expressions. Use this to implement `div-poly`, which takes two polys as arguments and returns a list of the quotient and remainder polys.

```
1   (define (div-terms L1 L2)
2     (if (empty-termlist? L1)
3         (list (the-empty-termlist) (the-empty-termlist))
4         (let ((t1 (first-term L1))
5               (t2 (first-term L2)))
6           (if (> (order t2) (order t1))
7               (list (the-empty-termlist) L1)
8               (let ((new-c (div (coeff t1) (coeff t2)))
9                     (new-o (- (order t1) (order t2))))
10                (let ((rest-of-result
11                       ;; <compute rest of result recursively>
12                       ))
13                  ;; <form complete result>
14                  ))))))
```

3.104.2   Answer

```
1   (define (div-terms L1 L2)
2     ;; termlist, termlist -> list of two termlists
3     ;; returns quotient terms and remainder terms
4     (if (empty-termlist? L1)
5         (list (the-empty-termlist) (the-empty-termlist))
6         (let ((t1 (first-term L1))
7               (t2 (first-term L2)))
8           (if (> (order t2) (order t1))
9               (list (the-empty-termlist) L1)
10              (let* ((new-term (make-sparse-term (- (order t1) (order t2))
11                                                 (div (coeff t1) (coeff
                                                   ↪   t2))))
12                     (rest-of-result
13                      (div-terms (sub-terms L1
14                                            (mul-terms (list new-term)
15                                                       L2))
16                                 L2)))
17                (list (adjoin-term new-term
18                                   (car rest-of-result))
19                      (cadr rest-of-result)))))))
20  (define (div-poly p1 p2)
```

313

```
21    ;; poly, poly -> list of 2 polys
22    ;; returns quotient poly and remainder poly
23    (if (same-variable? (variable p1) (variable p2))
24        (let* ((result (div-terms (terms p1) (terms p2)))
25               (quotient (car result))
26               (remainder (cadr result)))
27          (list (make-poly (variable p1) quotient)
28                (make-poly (variable p1) remainder)))
29        (error "Polys not in same var: DIV-POLY" (list p1 p2))))
30 (put 'div '(polynomial polynomial)
31       (lambda (p1 p2)
32         (let* ((result (div-poly p1 p2))
33                (quotient (car result))
34                (remainder (cadr result)))
35           (list (tag quotient)
36                 (tag remainder)))))
```

```
1   (load "mattcheck2.scm")
2   <<op-table>>
3   <<type-table>>
4   <<sqrt>>
5   <<echo>>
6   <<smarter-type-tagging>>
7   <<attach-tag-fixed>>
8   <<list-indices>>
9   <<tower-predicates>>
10  <<apply-generic-drop>>
11  <<generic-arithmetic-complex-parts>>
12  <<real-package>>
13  <<eqzero?>>
14  <<real-package>>
15  <<basic-tower-coercions>>
16
17  <<polynomial-sparse-dense-top>>
18  <<polynomial-division>>
19  <<polynomial-sparse-dense-bottom>>
20
21  <<generic-negation>>
22
23  (install-rational-package)
24  (install-rectangular-package)
25  (install-polar-package)
26  (install-complex-package)
27  (install-scheme-number-package)
28  (install-real-package)
29  (install-equ?)
30  (install-=zero?)
31  (install-polynomial-package)
```

```
32   (install-negation)

33

34   <<gen-math-tests>>

35

36   (mattcheck "raise success"
37             (raise 78)
38             (make-rational 78 1))
39   (mattcheck "raise fail"
40             (raise (make-complex-from-mag-ang 78 64))
41             #f)
42   (mattcheck "apply-generic 1"
43             (add 1 (make-rational 3 5))
44             (make-rational 8 5))
45   (mattcheck "apply-generic 2"
46             (add (make-real 2)
47                  (make-complex-from-mag-ang 3 5))
48             (make-complex-from-real-imag 2.850986556389679
             ↪   -2.8767728239894153))
49   (mattcheck "drop 1"
50             (drop (make-complex-from-real-imag 5 0))
51             5)
52   (mattcheck "drop 2"
53             (drop (make-real 5))
54             5)
55   (mattcheck "drop 3"
56             (drop (make-rational 5 1))
57             5)
58   (mattcheck "complex with nested rational"
59             (make-complex-from-real-imag 2 (make-rational 3 5))
60             (cons 'complex (cons 'rectangular (cons 2 (cons 'rational
             ↪   (cons 3 5))))))
61   (mattcheck "math on complex with nested rational"
62             (add (make-complex-from-real-imag 2 (make-rational 3 5))
63                  (make-complex-from-mag-ang (make-rational 3 5) 2))
64             (cons 'complex (cons 'rectangular
65                                 (cons (cons 'rational (cons
                                 ↪   8.751559490358574 5))
66                                      (cons 'rational (cons
                                      ↪   28.639461402385223 25))))))
67   (mattcheck "polynomial = zero"
68             (=zero? (make-polynomial 'x '((0 0)))))
69   (mattcheck "polynomial ≠ zero" #f
70             (=zero? (make-polynomial 'x '((1 1)(0 1)))))

71

72   (let ((p1 '((100 1)(2 2)(0 1)))
73         (p2 (list (list 7 (make-polynomial 'x '((3 2))))))
74         (p3 '((100 1)(2 1)(0 3)))
75         (p4 '((100 -1)(2 -2)(0 -1)))
```

315

```
76          (p5 '(8 2 5 -3 -7))
77          (p7 '((1 1)(0 1)))
78          (p6 '((3 1)(0 1)))
79          (p66 '(1 0 0 1))
80          (p77 '(1 1)))
81      (mattcheck "polynomial addition"
82                 (add (make-polynomial 'x p1)
83                      (make-polynomial 'x p1))
84                 '(polynomial x (100 2)(2 4)(0 2)))
85      (mattcheck "polynomial multiplication"
86                 (mul (make-polynomial 'x p1)
87                      (make-polynomial 'x p1))
88                 '(polynomial x (200 1) (102 4) (100 2) (4 4) (2 4) (0 1)))
89      (mattcheck "nested polynomial addition"
90                 (add (make-polynomial 'x p2)
91                      (make-polynomial 'x p1))
92                 '(polynomial x (100 1) (7 (polynomial x (3 2))) (2 2) (0
            ↪  1)))
93      (mattcheck "nested polynomial mult"
94                 (mul (make-polynomial 'x p2)
95                      (make-polynomial 'x p1))
96                 '(polynomial x (107 (polynomial x (3 2))) (9 (polynomial x
            ↪  (3 4))) (7 (polynomial x (3 2)))))
97      (mattcheck "negation scheme-number"
98                 (neg 5) -5)
99      (mattcheck "negation rational"
100                (neg (make-rational 1 5))
101                (make-rational -1 5))
102     (mattcheck "negation rectangular complex"
103                (neg (make-complex-from-real-imag 3 7))
104                (make-complex-from-real-imag -3 -7))
105     (mattcheck "negation polar complex"
106                (neg (make-complex-from-mag-ang 3 7))
107                (make-complex-from-mag-ang -3 7))
108
109     (mattcheck "negation polynomials"
110                (neg (make-polynomial 'x p1))
111                (make-polynomial 'x p4))
112     (mattcheck "polynomial subtraction"
113                (sub (make-polynomial 'x p1)
114                     (make-polynomial 'x p3))
115                '(polynomial x 1 0 -2))
116     (mattcheck "polynomial division 1"
117                (div (make-polynomial 'x p6)
118                     (make-polynomial 'x p7))
119                '((polynomial x 1 -1 1) (polynomial x)))
120     (mattcheck "polynomial division 2"
121                (div (make-polynomial 'x p7)
```

```
122                        (make-polynomial 'x p6))
123                 '((polynomial x) (polynomial x 1 1)))
124       (mattcheck "polynomial division 3"
125                 (div (make-polynomial 'x p66)
126                      (make-polynomial 'x p77))
127                 '((polynomial x 1 -1 1) (polynomial x))))
```

```
(etc ... )
SUCCEED at polynomial division 1
SUCCEED at polynomial division 2
SUCCEED at polynomial division 3
```

3.105 Exercise 2.92: operating on polys in different variables

3.105.1 Question

By imposing an ordering on variables, extend the polynomial package so that addition and multiplication of polynomials works for polynomials in different variables. (This is not easy!)

3.105.2 Answer

Unfortunately, I've spent a lot of time on this without making progress. For some reason it's really screwed with my brain. I'm going to move on to Exercise 2.93

If I understand the previous section correctly, I need to change the polynomial procedures so that when given two polys which include similar terms, say x with coefficients in y and y with coefficients in x, I need to pick one variable to be priority, and convert the other expression to match that priority. I suppose this would also need to work with multiple layers of polynomials, say a poly of w with nested coefficients x^{yz} and y with w^{zy}.

I've also decided to upgrade my `adjoin-terms` procedures so that they can adjoin to the middle of lists, or attempt to add two coefficients of the same order.

I've *also* also decided to add polynomials to the type coercion tower. This will require putting a variable on the new polynomial, but it will be a poly of one term of the 0'th order, which isn't really a polynomial. I'll need to add a check, `trivial-poly?`, which will allow the procedures to ignore the variable of such a polynomial.

My earlier choice to not tag sparse and dense lists may be coming back to bite me, now that I'm nesting items in the coefficients.

In the process of reorganizing polynomials, I'll make a hash table, and find the terms in the poly which are not dependent on the top variable, and split them into their own keys in the table. So for any terms of order 0, if it's a poly

317

it'll get keyed to the variable of the poly, and if it's not it'll be keyed to the symbol 'ZEROTH.

```scheme
(define tower
  '(scheme-number rational real complex polynomial))

(define (raise x)
  (let ((t1 (type-tag x)))
    (let recurse ((ll tower))
      (let ((t2 (car ll)))
        (cond ((null? (cdr ll)) #f)
              ((equal? t1 t2)
               ((get-coercion t1 (cadr ll)) (contents x)))
              (else (recurse (cdr ll))))))))

(put-coercion 'scheme-number 'rational
              (lambda(num)
                (make-rational num 1)))
<<eqnumber>>
(put-coercion 'rational 'real
              (lambda(num)
                (make-real
                 (exact->inexact (/ ((get 'numer '(rational)) num)
                                    ((get 'denom '(rational)) num))))))
(put-coercion 'real 'complex
              (lambda(num)
                (make-complex-from-real-imag
                 num 0)))
(put-coercion 'complex 'polynomial
              (lambda(num)
                ;; poly wants tagged numbers but (put-coercion) strips
                ;;   tags
                ;; also, might as well drop since it makes sense
                (let ((dropped-num (drop (attach-tag 'complex num))))
                  (make-polynomial 'DISCARD-ME
                                   (list (list 0 dropped-num))))))
```

```scheme
(use-srfis '(1))
(define (mem-index item ll)
  (list-index (lambda(x)(equal? item x))
              ll))
```

```scheme
(use-srfis '(1))
(define (install-polynomial-package) 'no)
;; internal procedures
;; representation of poly
(define (make-poly variable term-list)
```

```scheme
6      (cons variable
7            (convert-termlist-if-reasonable term-list)))
8    (define (variable p) (car p))
9    (define (terms p) (cdr p))
10   (define (variable? x) (symbol? x))
11   (define (same-variable? v1 v2)
12     (and (variable? v1) (variable? v2) (eq? v1 v2)))
13   (define (is-poly? list)
14     (eqv? 'polynomial (type-tag list)))

16   ;; representation of terms and term lists
17   (define (the-empty-termlist) '())
18   (define (rest-terms term-list) (cdr term-list))
19   (define (empty-termlist? term-list) (null? term-list))
20   (define (order term) (first term))
21   (define (coeff term) (second term))
22   (define (make-sparse-term order coeff)
23     (list order coeff))
24   (define (sparse-termlist? term-list)
25     (let ((t (car term-list)))
26       (cond ((and (pair? t) ;; looks like a sparse list
27                   (number? (first t))
28                   (or (pair? (second t)) ;; or sparse with nested poly
29                       (number? (second t))
30                       (empty-termlist? (second t)))) #t)
31             ((or (number? t) ;; looks dense
32                  (pair? t)
33                  (empty-termlist? term-list)) #f) ;; or dense with a
                  ↪  nested poly
34             (else (error "badly formed term: SPARSE-TERMLIST?" t
                  ↪  term-list)))))
35   ;; term and termlist procedures
36   (define (should-be-sparse? term-list)
37     ;; sparse or dense term-list -> bool
38     ;; if more than half of term-list is zeroes, return true
39     ;; reasoning:
40     ;; - number of elements used for sparse list is (non-zero elements * 2)
41     ;; - number of elements used for dense list is (highest order)
42     ;; also, if term-list is 1 non-number element, make a sparse term for
       ↪  clarity
43     (define (count-zeroes term-list)
44       ;; term-list -> (highest order, zeroes)
45       (let iter ((length 0)
46                  (zeroes 0)
47                  (TL (if (sparse-termlist? term-list)
48                          (sparse-to-dense term-list)
49                          term-list)))
50         (if (empty-termlist? TL)
```

```
51            (list zeroes length)
52            (iter (1+ length)
53                  (if (=zero? (car TL))
54                      (1+ zeroes)
55                      zeroes)
56                  (cdr TL)))))
57    (let ((stats (if (sparse-termlist? term-list) ;; do appropriate
   ↪    measurement
58                     (list (- (1+ (order (sparse-first-term term-list)))
59                              (length term-list))
60                           (1+ (order (sparse-first-term term-list))))
61                     (count-zeroes term-list))))
62      (let ((zero-elements (first stats))
63            (highest-order (second stats)))
64        (or (and (= 1 (length term-list))
65                 (not (number? (first-term term-list))))
66            (> (/ zero-elements highest-order)
67               (/ 1 2))))))
68  (define (convert-termlist-if-reasonable term-list)
69    (if (empty-termlist? term-list)
70        term-list
71        (cond ((sparse-termlist? term-list)
72               (let ((tl (sort-sparse-terms term-list)))
73                 (if (should-be-sparse? tl)
74                     tl
75                     (sparse-to-dense tl))))
76              ((should-be-sparse? term-list)
77               (dense-to-sparse term-list))
78              (else term-list))))
79  (define (sparse-to-dense term-list)
80    (let iter ((old-list (reverse term-list))
81               (new-list (the-empty-termlist)))
82      (if (empty-termlist? old-list)
83          new-list
84          (iter (rest-terms old-list)
85                (adjoin-term-dense (sparse-first-term old-list)
86                                   new-list)))))
87  (define (dense-to-sparse term-list)
88    (let iter ((old-list (reverse term-list))
89               (old-order 0)
90               (new-list (the-empty-termlist)))
91      (if (empty-termlist? old-list)
92          new-list
93          (iter (rest-terms old-list)
94                (1+ old-order)
95                (adjoin-term-sparse (make-sparse-term old-order
```

```
                                                    (coeff (dense-first-↵
                                                 ↪    term
                                                 ↪    old-list)))
                                    new-list)))))
(define (dense-termlist-order term-list)
  (1- (length term-list)))
(define (dense-first-term term-list)
  (make-sparse-term (dense-termlist-order term-list)
                    (car term-list)))
(define (sparse-first-term term-list)
  (car term-list))
(define (first-term term-list)
  (if (sparse-termlist? term-list)
      (sparse-first-term term-list)
      (dense-first-term term-list)))
(define (adjoin-term-sparse term term-list)
  ;; if term is higher order, cons to list
  ;; if equal, replace or add
  ;; if term is lower order than top of list, try to insert in the middle
  (if (empty-termlist? term-list)
      (cons term term-list)
      (let ((top-order (order (sparse-first-term term-list))))
        (cond ((≤ (1+ top-order) (order term))
               (cons term term-list))
              ((= top-order (order term))
               (if (=zero? (coeff (sparse-first-term term-list)))
                   ;; replace if it's zero
                   (cons term (rest-terms term-list))
                   ;; and add if it's not
                   (let ((result (add-sparse-term-to-sparse-term term
                                                                 (spars↵
                                                               ↪   e-↵
                                                               ↪   fi↵
                                                               ↪   rs↵
                                                               ↪   t-↵
                                                               ↪   te↵
                                                               ↪   rm
                                                               ↪   te↵
                                                               ↪   rm↵
                                                               ↪   -l↵
                                                               ↪   is↵
                                                               ↪   t)↵
                                                               ↪   )))
                     (cons result
                           (rest-terms term-list)))))
              (else (adjoin-term-sparse (sparse-first-term term-list)
                                        (adjoin-term-sparse term
```

```scheme
                                                      (rest-terms
                                                    ↪  term-li↲
                                                    ↪  st)))))↲
                                                    ↪  )))
(define (adjoin-term-dense term term-list)
  ;; if term is lower order than top of list, try to insert in the middle
  (let ((top-order (dense-termlist-order term-list)))
    (cond ((= (1+ top-order) (order term))
           ;; if term is 1 order above top of list, adjoin
           (cons (coeff term) term-list))
          ;; if term shares order with top of list,
          ((= top-order (order term))
           (if (=zero? (coeff (dense-first-term term-list)))
               ;; replace if it's zero
               (cons (coeff term)
                     (rest-terms term-list))
               ;; and add if it's not
               (let ((result (add-sparse-term-to-sparse-term term
                                                             (dense-firs↲
                                                           ↪  t-term
                                                           ↪  term-li↲
                                                           ↪  st))))
                 (cons (coeff result)
                       (rest-terms term-list)))))
          ((< top-order (order term))
           (adjoin-term-dense term (cons 0 term-list)))
          ;; if term is more than 1 order above, adjoin 0 and recurse
          (else (adjoin-term-dense (dense-first-term term-list)
                                   (adjoin-term-dense term
                                                      (rest-terms term-l↲
                                                    ↪  ist)))))))

(define (adjoin-term term term-list)
  (if (=zero? (coeff term))
      term-list
      (if (or (empty-termlist? term-list)
              (sparse-termlist? term-list))
          (adjoin-term-sparse term term-list)
          (adjoin-term-dense term term-list))))
(define (merge-terms t1 t2)
  (cond ((empty-termlist? t2)
         t1)
        ((empty-termlist? t1)
         t2)
        ((⩾ (order (first-term t1)) (order (first-term t2)))
         (adjoin-term (first-term t1)
                      (merge-terms (rest-terms t1) t2)))
        (else (adjoin-term (first-term t2)
```

```
170                              (merge-terms t1 (rest-terms t2))))))
171  ; ;; this second version is way simpler and way slower.
172  ; (define (merge-terms-abstracted t1 t2)
173  ;    (if (empty-termlist? t2)
174  ;        t1
175  ;        (merge-terms-abstracted (adjoin-term (first-term t2) t1)
176  ;                                    (rest-terms t2))))
177  (define (map-terms f tl)
178    ;; f is called (f order coeff)
179    (let iter ((result '())
180               (term-list tl))
181      (if (empty-termlist? term-list)
182          (reverse result)
183          (iter (cons (let ((t (first-term term-list)))
184                        (f (order t) (coeff t)))
185                      result)
186                (rest-terms term-list)))))
187  (define (map2-terms f tl1 tl2)
188    (let iter ((result '())
189               (term-list1 tl1)
190               (term-list2 tl2))
191      (if (or (empty-termlist? term-list1)
192              (empty-termlist? term-list2))
193          (reverse result)
194          (iter (cons (let ((t1 (first-term term-list))
195                            (t2 (first-term term-list)))
196                        (f (order t1) (coeff t1)
197                           (order t2) (coeff t2)))
198                      result)
199                (rest-terms term-list1)
200                (rest-terms term-list2)))))
201  (define (equ?-terms t1 t2)
202    (and (= (order t1)
203            (order t2))
204         (equ? (coeff t1)
205               (coeff t2))))
206  (define (sort-sparse-terms term-list)
207    (sort term-list (lambda(t1 t2)
208                      (> (order t1)
209                         (order t2)))))
210  (define (add-sparse-term-to-sparse-term t1 t2)
211    (make-sparse-term (order t1)
212                      (add (coeff t1) (coeff t2))))
213  (define (add-terms L1 L2)
214    (cond ((empty-termlist? L1) L2)
215          ((empty-termlist? L2) L1)
216          (else
217           (let ((t1 (first-term L1))
```

```
218                   (t2 (first-term L2)))
219               (cond ((> (order t1) (order t2))
220                      (adjoin-term
221                       t1 (add-terms (rest-terms L1) L2)))
222                     ((< (order t1) (order t2))
223                      (adjoin-term
224                       t2 (add-terms L1 (rest-terms L2))))
225                     (else
226                      (adjoin-term
227                       (add-sparse-term-to-sparse-term t1 t2)
228                       (add-terms (rest-terms L1)
229                                  (rest-terms L2)))))))))))
230  (define (mul-terms L1 L2)
231    (if (empty-termlist? L1)
232        (the-empty-termlist)
233        (add-terms (mul-term-by-all-terms (first-term L1) L2)
234                   (mul-terms (rest-terms L1) L2))))
235  (define (terms-=zero? L)
236    (or (empty-termlist? L)
237        (and (=zero? (coeff (first-term L)))
238             (terms-=zero? (rest-terms L)))))
239  (define (neg-terms TL)
240    (let ((negterm (if (sparse-termlist? TL)
241                       (lambda(t)
242                         (make-sparse-term (order t) (neg (coeff t))))
243                       (lambda(t)
244                         (neg (coeff t))))))
245      (map negterm TL)))
246  (define (sub-terms T1 T2)
247    (add-terms T1 (neg-terms T2)))
248  (define (div-terms L1 L2)
249    ;; termlist, termlist -> list of two termlists
250    ;; returns quotient terms and remainder terms
251    (if (empty-termlist? L1)
252        (list (the-empty-termlist) (the-empty-termlist))
253        (let ((t1 (first-term L1))
254              (t2 (first-term L2)))
255          (if (> (order t2) (order t1))
256              (list (the-empty-termlist) L1)
257              (let* ((new-term (make-sparse-term (- (order t1) (order t2))
258                                                 (div (coeff t1) (coeff
259                                                  ↪   t2))))
                     (rest-of-result
260                      (div-terms (sub-terms L1
261                                           (mul-terms (list new-term)
262                                                      L2))
263                                 L2)))
264                (list (adjoin-term new-term
```

324

```scheme
265                                       (car rest-of-result))
266                         (cadr rest-of-result)))))))
267  ;; polynomial procedures
268  (define (trivial-poly? poly)
269    (and (= 1 (length (terms poly)))
270         (= 0 (order (first-term (terms poly))))))
271  ;;; TODO: get working once cross-variable comparisons can be done
272  (put 'project 'polynomial
273       (lambda (p)
274         (if (not (trivial-poly? p))
275             #f
276             (coeff (first-term (terms p))))))
277  (define (equ?-poly p q)
278    (cond ((and (trivial-poly? p)
279                (trivial-poly? q))
280           (equ? (coeff (first-term (terms p)))
281                 (coeff (first-term (terms q)))))
282          ((same-variable? (variable p) (variable q))
283           (let iter ((term-list1 (reverse tl1))
284                      (term-list2 (reverse tl2)))
285             (cond ((and (empty-termlist? term-list1)
286                         (empty-termlist? term-list2))
287                    #t)
288                   ((or (empty-termlist? term-list1)
289                        (empty-termlist? term-list2))
290                    #f)
291                   ((equ?-terms (first-term term-list1)
292                                (first-term term-list2))
293                    (iter (rest-terms term-list1)
294                          (rest-terms term-list2)))
295                   (else #f))))
296          (else (error "different vars not supported yet : EQU?-POLY" p
297          ↪  q))))
297  (put 'equ? '(polynomial polynomial)
298       equ?-poly)
299  (define (make-variable-priority-list poly)
300    (cons (variable poly)
301          (let rec ((term-list (terms poly)))
302            (if (empty-termlist? term-list)
303                '()
304                (let* ((t (first-term term-list))
305                       (c (coeff t))
306                       (o (order t)))
307                  (cond ((is-poly? c)
308                         (cons (variable (contents c))
309                               (append (rec (rest-terms term-list))
310                                       (rec (terms (contents c))))))
311                        (else (rec (rest-terms term-list)))))))))
```

```scheme
312  (define (merge-priority-lists l1 l2)
313    (if (null? l2)
314        l1
315        (let rec ((to-check l1)
316                  (checking (car l2)))
317          (cond ((null? to-check)
318                 (merge-priority-lists (append l1 (list checking))
319                                       (cdr l2)))
320                ((same-variable? checking (car to-check))
321                 (merge-priority-lists l1
322                                       (cdr l2)))
323                (else (rec (cdr to-check) checking))))))
324  (define (poly-single-var? wantedvar poly)
325    (and (same-variable? (variable poly) wantedvar)
326         (let rec ((term-list (terms poly)))
327           (if (empty-termlist? term-list)
328               #t
329               (let ((c (coeff (first-term term-list))))
330                 (if (not (is-poly? c))
331                     (rec (rest-terms term-list))
332                     (and (poly-single-var? wantedvar c)
333                          (rec (rest-terms term-list)))))))))
334  ;;; Example of an inverted poly:
335  ;;; (poly x (order 4, coeff (poly y (5 6)))
336  ;;;         (order 3, coeff (poly y (1 2)))
337  ;;;         (order 0, coeff 7))
338  ;;; (poly y (order 5, coeff (poly x (4 6)))
339  ;;;         (order 1, coeff (poly x (3 2)))
340  ;;;         (order 0, coeff 7))
341  ;;; (make-poly sub-var (list (make-sparse-term subO (make-polynomial
     ↪  super-var (list (make-sparse-term superO subC))))))
342  ;;; (poly x (1, (poly y (2, (poly z (3, 4))))))
343  ;;; (poly x (1, (poly z (3, (poly y (2, 4))))))
344  ;;; (poly z (3, (poly x (1, (poly y (2, 4))))))
345  ;;; (poly z (3, (poly y (2, (poly x (1, 4))))))
346
347  ;; This isn't good enough, it doesn't invert the poly and also it needs
     ↪  to
348  ;; produce multiple terms for one term with a poly with multiple terms
349  ;(define (invert-term parent-var parent-order term)
350  ;  (let* ((child-order (order term))
351  ;        (child-coeff (coeff term)))
352  ;    (make-sparse-term child-order
353  ;                      (make-polynomial parent-var
354  ;                                       (list (make-sparse-term
     ↪  parent-order
355  ;
     ↪  child-coeff))))))
```

```scheme
356  (define (pad-term-if-needed term parent-priority-list)
357    ;; term with tagged poly, priority list -> term
358    ;; parent-priority-list starts with the variable this term's order is
     ↪  in
359    ;; term: term with tagged poly inside
360
361    ;; pad term so that cross-variable functions will have equal structures
362    ;; to thoughtlessly work from. I.e. if your list is '(a b c d e) and
363    ;; your term is (5 (poly d (2 1))), we'll return
364    ;; '(5 (poly b (poly c (poly d (2 1))))) (nested polys representing
     ↪  dense terms)
365    (let* ((parent-var (car parent-priority-list))
366           (child-var (variable (contents (coeff term))))
367           (pri (member parent-var parent-priority-list))
368           (difference (mem-index child-var pri)))
369      (if (= difference 1)
370          term
371          (let* ((rest-priorities (cdr pri))
372                 (next-var (car rest-priorities)))
373            (make-sparse-term (order term)
374                              (make-polynomial next-var
375                                               (list (pad-term-if-needed
                                                   ↪  (make-sparse-term 0
                                                   ↪  (coeff term))
376                                                    rest-priorities))))))))
```

```scheme
377  (define (pad-poly-if-needed poly parent-priority-list)
378    ;; untagged poly -> untagged poly
379    (let* ((child-var (variable poly))
380           (difference (mem-index child-var parent-priority-list)))
381      (if (> difference 0)
382          (make-poly (car parent-priority-list)
383                     (list (pad-term-if-needed (make-sparse-term 0 (tag
                             ↪ poly))
384                                                parent-priority-list))))))
385  (define (make-empty-poly-in var)
386    (if (pair? var)
387        (make-poly (car var)
388                   (list (make-sparse-term
389                          0 (tag (make-empty-poly-in
390                                  (if (= 2 (length var))
391                                      (cadr var)
392                                      (cdr var)))))))
393        (make-poly var (the-empty-termlist))))
394  (define (priority-depth var priority-list)
395    ;; return how far in a list a var is
396    ;; 0 is head, 2 means two vars ahead, etc
397    (mem-index var priority-list))
398  (define (priority-diff var1 var2 priority-list)
399    ;; return how far ahead var2 is from var1
400    ;; so a child correctly nested in a parent is 1
401    ;; if vars need to be swapped, this will be negative
402    (let ((d1 (mem-index var1 priority-list))
403          (d2 (mem-index var2 priority-list)))
404      (- d2 d1)))
405  (define (invert-nested-poly-terms superVar superO term-list)
406    ;;  -> term-list
407    (let iter ((tl term-list)
408               (done (the-empty-termlist)))
409      (if (empty-termlist? tl)
410          done
411          (let* ((t (first-term tl))
412                 (new-t (make-sparse-term (order t)
413                                          (make-polynomial superVar
414                                                           (make-sparse-te↵
                                                            ↪ rm
                                                            ↪ superO
```

```scheme
415
              (iter (rest-terms tl)
                    (adjoin-term new-t done))))))
  (define (poly-has-no-children? p)
    (let iter ((tl (terms p)))
      (cond ((empty-termlist? tl) #t)
            ((and (pair? (coeff (first-term tl)))
                  (equal? 'polynomial (type-tag (coeff (first-term tl)))))
             #f)
            (else (iter (rest-terms tl))))))
  ;; BIG PROBLEM: I NEED TO WORK OUT ORDERING OF TERMS VERSUS ORDERING OF
  ;;   POLY NESTING
  ;; i.e.
  ;(poly a (0
  ;        (poly b (1 2)
  ;               (0
  ;               (poly c (3 4))))))
  ;(poly a (0 (poly b (1 2)))
  ;        (poly c (3 4)))
  ;; Maybe the only good choice is to have polys strictly nest linearly
  (define (reorganize-poly-if-needed poly parent-priority-list)
    ;; untagged poly, pri list -> untagged poly
    ;; pri list should start with the variable you want the poly in,
```

```scheme
436    ;; and contain all vars that the poly contains
437   (define (promote p target-var)
438     (define (p-rec term-list)
439       (map-terms (lambda(t)
440                    ())
441                  term-list)))
442     ;; descend into children until findng desired var, then ascend
443     ;; to bring it back to the parent while rearranging as needed
444     (receive (cool uncool)
445        (partition (lambda(t)
446                     (let ((c (coeff t)))
447                     (and (pair? c)
448                          (equal? 'polynomial (type-tag c))
449                          (same-variable? target-var
450                                          (variable (contents c))))))
451                  (let ((tl (terms p)))
452                    (if (sparse-termlist? tl)
453                        tl
454                        (dense-to-sparse tl))))
455      (cond ((poly-has-no-children? p) p) ;;dead end
456            ((null? cool) ;; var is below but not here yet, keep bubbling
457             )
458            (else (make-poly target-var
459                             (let rec ((tl cool))
460                               (if (empty-termlist? tl)
461                                   (the-empty-termlist)
462                                   (let* ((t (first-term tl))
463                                          (o (order t))
464                                          (c (coeff c))
465                                          (child-poly (contents c)))
466                                     (merge-termlists
                                     ↪  (invert-nested-poly-terms
                                     ↪  (variable p) o
```

```
467
468                                                      (rec (rest-terms
                                                     ↪ tl)))))))))))
469    (let rec ((p poly)
470             (pri parent-priority-list))
471      (if (null? pri)
472          p ;; give up
473          (let* ((top-var (car pri))
```

```scheme
                   (var (variable p))
                   (tl (terms p))
                   (child-descendants (make-variable-priority-list p))
                   (diff (priority-diff top-var var parent-priority-list)))
              (cond ((equal? child-descendants parent-priority-list)
                      p) ;; sort done
                     ((not (member top-var child-descendants))
                      ;; we need to pad for this var
                      (make-poly top-var (tag (rec p (cdr pri)))))
                     ((= 0 diff)
                      ('todo)))))))
;(define (check-terms term-list)
;   (let iter ((remaining term-list)
;              (done (the-empty-termlist)))
;     (if (empty-termlist? remaining)
;         done
;         (let* ((t (first-term remaining))
;                (c (coeff t))
;                (o (order t))
;                (new-t (make-sparse-term o
;                                          (if (is-poly? c)
;                                              (tag
;   (reorganize-poly-if-needed (contents c) parent-priority-list))
;                                              c))))
;           (iter (rest-terms remaining)
;                 (adjoin-term new-t done))))))
;(define (invert-this-poly poly new-var)
;   (let ((var (variable poly))
;         (term-list (terms poly)))
;     (let iter ((remaining term-list)
;                (done (the-empty-termlist)))
;       (if (empty-termlist? remaining)
;           (make-poly new-var done)
;           (let* ((t (first-term remaining))
;                  (o (order t))
;                  (c (coeff t)))
;             (if (is-poly? c)
;                 (let ((child-var (variable (contents c)))
;                       (child-terms (terms (contents c))))
;                   (cond ((> o 0) ;; if order is not 0, an inverted poly
;   is needed
;                          (iter (rest-terms remaining)
;                                (merge-termlists done
;   (invert-nested-poly-terms new-var o child-terms))))
;                         ((same-variable? var child-var)
;                          (error "This shouldn't happen :
;   INVERT-THIS-POLY :" remaining done))
```

```
517  ;                                          ;(iter (merge-termlists
   ↪  child-terms (rest-terms remaining))
518  ;                                          ;        done))
519  ;                          ((same-variable? new-var child-var)
520  ;                           (iter (rest-terms remaining)
521  ;                                 (merge-termlists done (check-terms
   ↪  child-terms))))
522  ;                          (else (error "WHAT TO HECK : INVERT-THIS-POLY
   ↪  :" remaining done))))
523  ;                 (let ((new-t (make-sparse-term 0
524  ;                                              (make-polynomial var
525  ;
   ↪  (make-sparse-term o
526  ;
   ↪           c)))))
527  ;                   (iter (rest-terms remaining)
528  ;                         (adjoin-term new-t done)))))))))
529  ;(let* ((top-var (car parent-priority-list))
530  ;        (var (variable poly))
531  ;        (tl (terms poly))
532  ;        (child-descendants (make-variable-priority-list poly))
533  ;        (diff (priority-diff top-var var parent-priority-list)))
534  ;  (cond ((= 0 diff)
535  ;          ;; if poly and top-var are the same, then keep terms how they
   ↪  are
536  ;          ;; except for terms that have poly coeffs, which need to be
   ↪  checked
537  ;          (make-poly top-var (check-terms tl)))
538  ;          ((< 0 diff)
539  ;           (let ((reorganized-terms (check-terms tl)))
540  ;             (pad-poly-if-needed (make-poly var reorganized-terms)
541  ;                                 parent-priority-list)))
542  ;          ;; if var is 1 step of priority below in the list than top-var,
   ↪  then
543  ;          ;; swap its terms with top-var terms
544  ;          ((= 1 diff)
545  ;           (pad-poly-if-needed (invert-this-poly poly top-var)
546  ;                               parent-priority-list))
547  ;          ;; if var is >1 steps of priority below top-var, and top-var is
   ↪  known
548  ;          ;; to be in the child's children, then reorganize the child
   ↪  poly's
549  ;          ;; terms and try again
550  ;          ((< 1 diff)
551  ;           (if (member top-var child-descendants)
552  ;               (let ((reorganized-terms (check-terms tl)))
553  ;                 (reorganize-poly-if-needed (make-poly var
   ↪  reorganized-terms)
```

333

```scheme
554  ;                                          parent-priority-list))
555  ;              (error "expected child poly not found? :
 ↪   REORGANIZE-POLY-IF-NEEDED :"
556  ;                     parent-priority-list child-descendants)))
557  ;         (else "what the SCALLOP" child-descendants diff))))
558  (define (operate-on-polys f p1 p2)
559    ;; f is a function to operate on term-lists
560    (cond ((same-variable? (variable p1) (variable p2))
561           (make-poly (variable p1)
562                      (f (terms p1) (terms p2))))
563          ((trivial-poly? p1)
564           (make-poly (variable p2)
565                      (f (terms p1) (terms p2))))
566          ((trivial-poly? p2)
567           (make-poly (variable p1)
568                      (f (terms p1) (terms p2))))
569          (((get '=zero? 'polynomial) p1)
570           p2)
571          (((get '=zero? 'polynomial) p2)
572           p1)
573          (else
574           (let* ((priorities (merge-priority-lists
                     ↪  (make-variable-priority-list p1)
575                                               (make-variable-priorit⌋
                                                 ↪  y-list
                                                 ↪  p2)))
576                  (p1r ((reorganize-poly-if-needed p1 priorities)))
577                  (p2r ((reorganize-poly-if-needed p2 priorities))))
578             (make-poly (variable p1r)
579                        (f (terms p1r) (terms p2r)))))))
580  (define (add-poly p1 p2)
581    (define triv
582      ;; if a poly is trivial, return the opposite variable
583      (cond ((trivial-poly? p1)
584             (variable p2))
585            ((trivial-poly? p2)
586             (variable p1))
587            (else #f)))
588    (cond ((or (same-variable? (variable p1) (variable p2))
589               triv)
590           (make-poly (if triv
591                          triv
592                          (variable p1))
593                      (add-terms (terms p1) (terms p2))))
594          (else
595           (let ((p2r ((reorganize-poly-if-needed p2
```

```
596                                                        (make-variable-priority-↵
                                                     ↪   list
                                                     ↪   p1)))))
597               (make-poly (variable p1)
598                          (add-terms (terms p1) (terms p2r)))))))
599   (define (mul-poly p1 p2)
600     (define triv
601       ;; if a poly is trivial, return the opposite variable
602       (cond ((trivial-poly? p1)
603              (variable p2))
604             ((trivial-poly? p2)
605              (variable p1))
606             (else #f)))
607     (cond ((or (same-variable? (variable p1) (variable p2))
608                triv)
609            (make-poly (if triv
610                           triv
611                           (variable p1))
612                       (mul-terms (terms p1) (terms p2))))
613           (else (error "Polys not in same var: MUL-POLY" (list p1 p2)))))
614
615   ;; interface to rest of the system
616   (define (tag p) (attach-tag 'polynomial p))
617   (put 'add '(polynomial polynomial)
618        (lambda (p1 p2) (tag (add-poly p1 p2))))
619   (put 'mul '(polynomial polynomial)
620        (lambda (p1 p2) (tag (mul-poly p1 p2))))
621   (put 'make 'polynomial
622        (lambda (var terms) (tag (make-poly var terms))))
623   (define (if-1st-poly-make-2nd-poly t1 t2)
624     (if (and (eqv? 'polynomial (type-tag t1))
625              (not (eqv? 'polynomial (type-tag t2))))
626         (tag (make-poly (variable (contents t1)) (list t2)))
627         t2))
628   (define (mul-term-by-all-terms t1 L)
629     (if (empty-termlist? L)
630         (the-empty-termlist)
631         (let* ((t2 (first-term L)) ;; If one coeff is a poly, make the
                ↪   other a poly
632                (t1-coeff (if-1st-poly-make-2nd-poly (coeff t2) (coeff t1)))
633                (t2-coeff (if-1st-poly-make-2nd-poly (coeff t1) (coeff
                ↪   t2))))
634
635           (adjoin-term
636            (make-sparse-term (+ (order t1) (order t2))
637                              (mul t1-coeff t2-coeff))
638            (mul-term-by-all-terms t1 (rest-terms L))))))
639   (define (poly-=zero? p)
```

335

```
640    (terms-=zero? (terms p)))
641  (put '=zero? '(polynomial) poly-=zero?)
642  (define (neg-poly p)
643    (make-poly (variable p)
644              (neg-terms (terms p))))
645  (put 'neg '(polynomial)
646       (lambda (p1) (tag (neg-poly p1))))
647  (define (sub-poly p1 p2)
648    (add-poly p1 (neg-poly p2)))
649  (put 'sub '(polynomial polynomial)
650       (lambda (p1 p2) (tag (sub-poly p1 p2))))
651  (define (div-poly p1 p2)
652    ;; poly, poly -> list of 2 polys
653    ;; returns quotient poly and remainder poly
654    (if (same-variable? (variable p1) (variable p2))
655        (let* ((result (div-terms (terms p1) (terms p2)))
656               (quotient (car result))
657               (remainder (cadr result)))
658          (list (make-poly (variable p1) quotient)
659                (make-poly (variable p1) remainder)))
660        (error "Polys not in same var: DIV-POLY" (list p1 p2))))
661  (put 'div '(polynomial polynomial)
662       (lambda (p1 p2)
663         (let* ((result (div-poly p1 p2))
664                (quotient (car result))
665                (remainder (cadr result)))
666           (list (tag quotient)
667                 (tag remainder)))))
```

```
1  ;  'done)
2  (define (make-polynomial var terms)
3    ((get 'make 'polynomial) var terms))
```

```
1  (load "../mattcheck2.scm")
2  <<op-table>>
3  <<type-table>>
4  <<sqrt>>
5  <<echo>>
6  <<smarter-type-tagging>>
7  <<attach-tag-fixed>>
8  <<list-indices>>
9  <<tower-predicates>>
10 <<apply-generic-drop>>
11 <<generic-arithmetic-complex-parts>>
12 <<real-package>>
13 <<eqzero?>>
14 <<real-package>>
```

```
15   <<tower-coercions-polynomial>>

16

17   <<mem-index>>
18   <<polynomial-mixed-variables-top>>
19   <<polynomial-mixed-variables-bottom>>

20

21   <<generic-negation>>

22

23   (install-rational-package)
24   (install-rectangular-package)
25   (install-polar-package)
26   (install-complex-package)
27   (install-scheme-number-package)
28   (install-real-package)
29   (install-equ?)
30   (install-=zero?)
31   (install-polynomial-package)
32   (install-negation)

33

34   <<gen-math-tests>>

35

36   (mattcheck "raise success"
37             (raise 78)
38             (make-rational 78 1))
39   (mattcheck "raise fail"
40             (raise (make-polynomial 'x '(3 2)))
41             #f)
42   (mattcheck "apply-generic 1"
43             (add 1 (make-rational 3 5))
44             (make-rational 8 5))
45   (mattcheck "apply-generic 2"
46             (add (make-real 2)
47                  (make-complex-from-mag-ang 3 5))
48             (make-complex-from-real-imag 2.850986556389679
             ↪  -2.8767728239894153))
49   (mattcheck "drop 1"
50             (drop (make-complex-from-real-imag 5 0))
51             5)
52   (mattcheck "drop 2"
53             (drop (make-real 5))
54             5)
55   (mattcheck "drop 3"
56             (drop (make-rational 5 1))
57             5)
58   (mattcheck "complex with nested rational"
59             (make-complex-from-real-imag 2 (make-rational 3 5))
60             (cons 'complex (cons 'rectangular (cons 2 (cons 'rational
             ↪  (cons 3 5))))))
```

```
61  (mattcheck "math on complex with nested rational"
62            (add (make-complex-from-real-imag 2 (make-rational 3 5))
63                 (make-complex-from-mag-ang (make-rational 3 5) 2))
64            (cons 'complex (cons 'rectangular
65                                 (cons (cons 'rational (cons
                                      ↪ 8.751559490358574 5))
66                                       (cons 'rational (cons
                                            ↪ 28.639461402385223 25))))))
67  (mattcheck "polynomial = zero"
68            (=zero? (make-polynomial 'x '((0 0)))))
69  (mattcheck "polynomial ≠ zero" #f
70            (=zero? (make-polynomial 'x '((1 1)(0 1)))))
71
72  (define p1 '((100 1)(2 2)(0 1)))
73  (define p2 (list (list 7 (make-polynomial 'x '((3 2))))))
74  (define p3 '((100 1)(2 1)(0 3)))
75  (define p4 '((100 -1)(2 -2)(0 -1)))
76  (define p5 '(8 2 5 -3 -7))
77  (define p7 '((1 1)(0 1)))
78  (define p6 '((3 1)(0 1)))
79  (define p66 '(1 0 0 1))
80  (define p77 '(1 1))
81  (define p8 (make-poly 'a
82                        (list (list 9 (make-polynomial 'b p3))
83                              (list 8 (make-polynomial 'c p5))
84                              (list 1 (make-polynomial 'd (list (list 0
                                   ↪ (make-polynomial 'f '((3 2)))))))
85                              (list 0 (make-polynomial 'e (list 2 0))))))
86  (define p9 (make-poly 'a
87                        (list (list 9
88                                    (make-polynomial
89                                     'b (list (list 8 (make-polynomial
90                                                      'c (list (list 1
                                                          ↪ (make-polynomial
```

338

```
⌋
↪ 'd⌋
↪
↪
↪ ⌋
↪ ⌋
↪ (⌋
↪
↪ l⌋
↪ i⌋
↪ s⌋
↪ t⌋
↪
↪
↪ ⌋
↪ ⌋
↪ (⌋
↪
↪ l⌋
↪ i⌋
↪ s⌋
↪ t⌋
↪
↪
↪ ⌋
↪ |⌋
↪ 0⌋
↪
↪
↪ ⌋
↪ ⌋
↪ (⌋
↪
↪ m⌋
↪ a⌋
↪ k⌋
↪ e⌋
↪ -⌋
↪ p⌋
↪ o⌋
↪ l⌋
↪ y⌋
↪ n⌋
↪ o⌋
↪ m⌋
↪ i⌋
↪ a⌋
↪ l
```

341

↪ ⌋
↪ ⌋
↪ ⌋
↪ ⌋
↪ ⌋
↪ ⌋
↪ ⌋
↪ ⌋
↪ ⌋
↪ ⌋
↪ ⌋
↪ ⌋
↪ ⌋
↪ ⌋
↪ ⌋
↪ ⌋
↪ ⌋
↪ ⌋
↪ ⌋
↪ ⌋
↪ ⌋
↪ ⌋
↪ ⌋
↪
↪ ⌋
↪ 'e⌋
↪ ⌋
↪
↪ ⌋
↪ ⌋
↪ (⌋
↪ ⌋
↪ l⌋
↪ i⌋
↪ s⌋
↪ t⌋
↪ ⌋
↪
↪ ⌋
↪ ⌋
↪ 2⌋
↪ ⌋
↪
↪ ⌋
↪ ⌋
↪ 0⌋
↪ ⌋
↪)⌋
↪)⌋
↪)⌋
↪)⌋
↪)⌋
↪)⌋

```
93   (mattcheck "polynomial addition"
94              (add (make-polynomial 'x p1)
95                   (make-polynomial 'x p1))
96              '(polynomial x (100 2)(2 4)(0 2)))
97   (mattcheck "polynomial multiplication"
98              (mul (make-polynomial 'x p1)
99                   (make-polynomial 'x p1))
100             '(polynomial x (200 1) (102 4) (100 2) (4 4) (2 4) (0 1)))
101  (mattcheck "nested polynomial addition"
102             (add (make-polynomial 'x p2)
103                  (make-polynomial 'x p1))
104             '(polynomial x (100 1) (7 (polynomial x (3 2))) (2 2) (0 1)))
105  (mattcheck "nested polynomial mult"
106             (mul (make-polynomial 'x p2)
107                  (make-polynomial 'x p1))
108             '(polynomial x (107 (polynomial x (3 2))) (9 (polynomial x (3
     ↪  4))) (7 (polynomial x (3 2)))))
109  (mattcheck "negation scheme-number"
110             (neg 5) -5)
111  (mattcheck "negation rational"
112             (neg (make-rational 1 5))
113             (make-rational -1 5))
114  (mattcheck "negation rectangular complex"
115             (neg (make-complex-from-real-imag 3 7))
116             (make-complex-from-real-imag -3 -7))
117  (mattcheck "negation polar complex"
118             (neg (make-complex-from-mag-ang 3 7))
119             (make-complex-from-mag-ang -3 7))
120
121  (mattcheck "negation polynomials"
122             (neg (make-polynomial 'x p1))
123             (make-polynomial 'x p4))
124  (mattcheck "polynomial subtraction"
125             (sub (make-polynomial 'x p1)
126                  (make-polynomial 'x p3))
127             '(polynomial x 1 0 -2))
128  (mattcheck "polynomial division 1"
129             (div (make-polynomial 'x p6)
130                  (make-polynomial 'x p7))
131             '((polynomial x 1 -1 1) (polynomial x)))
132  (mattcheck "polynomial division 2"
133             (div (make-polynomial 'x p7)
134                  (make-polynomial 'x p6))
135             '((polynomial x) (polynomial x 1 1)))
136  (mattcheck "polynomial division 3"
137             (div (make-polynomial 'x p66)
138                  (make-polynomial 'x p77))
139             '((polynomial x 1 -1 1) (polynomial x)))
```

```
140
141   (mattcheck "poly projection good"
142             (project (make-polynomial 'useless '((0 7))))
143             7)
144   (mattcheck "poly projection bad"
145             (project (make-polynomial 'useless '((1 7))))
146             #f)
147   (mattcheck "raise to poly"
148             (raise (make-complex-from-mag-ang 78 64))
149             '(polynomial DISCARD-ME (complex polar 78 . 64)))
150   (mattcheck "reorganize-poly-if-needed"
151             (reorganize-poly-if-needed p8 '(f e d c b a)))
152   (mattcheck "polynomials in multiple variables 1"
153             (add (make-polynomial 'x p1)
154                  (make-polynomial 'y p1))
155             'unknown)
```

3.106 Exercise 2.93: extending `make-rational` for rational functions

3.106.1 Question

Modify the `rational-arithmetic` package to use generic operations, but change `make-rat` so that it does not attempt to reduce fractions to lowest terms. Test your system by calling `make-rational` on two polynomials to produce a rational function:

```
1   (define p1 (make-polynomial 'x '((2 1) (0 1))))
2   (define p2 (make-polynomial 'x '((3 1) (0 1))))
3   (define rf (make-rational p2 p1))
```

Now add `rf` to itself, using `add`. You will observe that this addition procedure does not reduce fractions to lowest terms.

3.106.2 Answer

`raise`-ing a rational function doesn't work, so I'll need a way around that.

```
1    (define tower
2      '(scheme-number rational real complex polynomial))
3
4    (define (raise x)
5      (let ((t1 (type-tag x)))
6        (let recurse ((ll tower))
7          (let ((t2 (car ll)))
8            (cond ((null? (cdr ll)) #f)
9                  ((equal? t1 t2)
10                  ((get-coercion t1 (cadr ll)) (contents x)))
```

```
11                    (else (recurse (cdr ll))))))))))

12

13    (define (raise-to x target)
14      (define current-type (type-tag x))
15      (define (try-coerce type-to-try)
16        (let ((up (get-coercion current-type type-to-try)))
17          (if up
18              (up (contents x))
19              #f)))
20      (let* ((tower-from-here (member current-type tower)))
21        (if (or (not tower-from-here)
22                (not (member target tower-from-here)))
23            #f ;; this raise can't happen
24            (let rec ((ll (cdr tower-from-here)))
25              (if (eqv? target (car ll))
26                  (try-coerce (car ll))
27                  (let ((try-harder
28                         (rec (cdr ll))))
29                    (if try-harder
30                        try-harder
31                        (try-coerce (car ll)))))))))

32

33    (put-coercion 'scheme-number 'rational
34                  (lambda(num)
35                    (make-rational num 1)))
36    <<eqnumber>>
37    (put-coercion 'rational 'real
38                  (lambda(num)
39                    (let ((result (div ((get 'numer '(rational)) num)
40                                       ((get 'denom '(rational)) num))))
41                      (if (number? result)
42                          (make-real (exact->inexact result))
43                          #f))))
44    (put-coercion 'real 'complex
45                  (lambda(num)
46                    (make-complex-from-real-imag
47                     num 0)))
48    (let ((raised-poly (lambda(num tag)
49                         (make-polynomial 'DISCARD-ME
50                                          (list (list 0 (attach-tag tag
       ↪  num)))))))
51      (put-coercion 'scheme-number 'polynomial
52                    (lambda(num)
53                      (raised-poly num 'scheme-number)))
54      (put-coercion 'rational 'polynomial
55                    (lambda(num)
56                      (raised-poly num 'rational)))
57      (put-coercion 'real 'polynomial
```

```
58                    (lambda(num)
59                      (raised-poly num 'real)))
60     (put-coercion 'complex 'polynomial
61                      (lambda(num)
62                        (raised-poly num 'complex))))
```

```
1   (use-srfis '(1))
2   ;; Version of apply-generic that tries to drop its type
3   ;; when returning
4   (define (apply-generic op . args)
5     (define types (map type-tag args))
6     (define (map-raise highest-type item)
7       (if (equal? (type-tag item) highest-type)
8           item
9           (let ((raised (raise-to item highest-type)))
10            (if (not raised)
11                (error "MAP-RAISE: could not raise" item highest-type)
12                (map-raise highest-type raised)))))
13
14    (drop
15     (let ((proc (get op types)))
16       (if proc ; is this already doable?
17           (apply proc (map contents args))
18                                            ; if not, try type coercion
19
20           (let* ((highest-type (apply latest-in-tower types))
21                  (map-raise-prime (lambda(item)(map-raise highest-type
                    ↪  item)))
22                  (raised (map map-raise-prime args)))
23             (if (< (length raised) 3)
24                 ;; short args
25                 (let ((f (get op (map type-tag raised))))
26                   (if (eqv? #f f)
27                       (error ↵
                        ↪  "APPLY-GENERIC: could not find procedure for types"↵
                        ↪  (map type-tag raised)
                        ↪  raised)
28                       (apply f (map contents raised))))
29                 ;; long args
30                 (reduce (lambda(elem previous)
31                           (apply-generic op previous elem))
32                         "ERROR" raised)))))))
```

```
1   (use-srfis '(1))
2   (define (install-polynomial-package) 'no)
3     ;; internal procedures
4     ;; representation of poly
```

345

```scheme
    (define (make-poly variable term-list)
      (cons variable
            (convert-termlist-if-reasonable term-list)))
    (define (convert-termlist-if-reasonable term-list)
      (if (empty-termlist? term-list)
          term-list
          (cond ((sparse-termlist? term-list)
                 (let ((tl (sort-sparse-terms term-list)))
                   (if (should-be-sparse? tl)
                       tl
                       (sparse-to-dense tl))))
                ((should-be-sparse? term-list)
                 (dense-to-sparse term-list))
                (else term-list))))
    (define (variable p) (car p))
    (define (terms p) (cdr p))
    (define (variable? x) (symbol? x))
    (define (same-variable? v1 v2)
      (and (variable? v1) (variable? v2) (eq? v1 v2)))

    ;; representation of terms and term lists
  (define (sparse-termlist? term-list)
    (let ((t (car term-list)))
      (cond ((and (pair? t) ;; looks like a sparse list
                  (number? (first t))
                  (or (pair? (second t)) ;; or sparse with nested poly
                      (number? (second t))
                      (empty-termlist? (second t)))) #t)
            ((or (number? t) ;; looks dense
                 (pair? t)
                 (empty-termlist? term-list)) #f) ;; or dense with a
                 ↪ nested poly
            (else (error "badly formed term: SPARSE-TERMLIST?" t
                 ↪ term-list)))))
  (define (sort-sparse-terms term-list)
    (sort term-list (lambda(t1 t2)
                      (> (order t1)
                         (order t2)))))
    (define (should-be-sparse? term-list)
      ;; sparse or dense term-list -> bool
      ;; if more than half of term-list is zeroes, return true
      ;; reasoning:
      ;; - number of elements used for sparse list is (non-zero elements *
      ↪  2)
      ;; - number of elements used for dense list is (highest order)
      (define (count-zeroes term-list)
        ;; term-list -> (highest order, zeroes)
        (let iter ((length 0)
```

```scheme
                       (zeroes 0)
                       (TL (if (sparse-termlist? term-list)
                               (sparse-to-dense term-list)
                               term-list)))
          (if (empty-termlist? TL)
              (list zeroes length)
              (iter (1+ length)
                    (if (=zero? (car TL))
                        (1+ zeroes)
                        zeroes)
                    (cdr TL)))))))
    (let ((stats (if (sparse-termlist? term-list) ;; do appropriate
      ↪   measurement
                     (list (- (1+ (order (sparse-first-term term-list)))
                              (length term-list))
                           (1+ (order (sparse-first-term term-list))))
                     (count-zeroes term-list))))
      (let ((zero-elements (first stats))
            (highest-order (second stats)))
        (> (/ zero-elements highest-order)
           (/ 1 2)))))))
(define (sparse-to-dense term-list)
  (let iter ((old-list (reverse term-list))
             (new-list (the-empty-termlist)))
    (if (empty-termlist? old-list)
        new-list
        (iter (rest-terms old-list)
              (adjoin-term-dense (sparse-first-term old-list)
                                 new-list)))))
(define (dense-to-sparse term-list)
  (let iter ((old-list (reverse term-list))
             (old-order 0)
             (new-list (the-empty-termlist)))
    (if (empty-termlist? old-list)
        new-list
        (iter (rest-terms old-list)
              (1+ old-order)
              (adjoin-term-sparse (make-sparse-term old-order
                                                    (coeff (dense-firs⌋
                                                      ↪   t-term
                                                      ↪   old-list)))
                                  new-list)))))
(define (make-sparse-term order coeff)
  (list order coeff))
(define (dense-termlist-order term-list)
  (1- (length term-list)))
(define (dense-first-term term-list)
  (make-sparse-term (dense-termlist-order term-list)
```

347

```scheme
                          (car term-list)))
    (define (sparse-first-term term-list)
      (car term-list))
    (define (first-term term-list)
      (if (sparse-termlist? term-list)
          (sparse-first-term term-list)
          (dense-first-term term-list)))
    (define (adjoin-term-sparse term term-list)
      (cons term term-list))
    (define (adjoin-term-dense term term-list)
      ;; if term is exactly 1 order above top of list, adjoin
      ;; if term is more than 1 order above, adjoin 0 and recurse
      (let ((top-order (dense-termlist-order term-list)))
        (cond ((= (1+ top-order) (order term))
                (cons (coeff term) term-list))
              ((≤ top-order (order term))
               (adjoin-term-dense term (cons 0 term-list)))
              (else (error ⌋
          ↪ "term is lower order than top of list: ADJOIN-TERM-DENSE"⌋
          ↪    term term-list
          ↪   top-order)))))

    ;; ;; I wrote the versions below which can insert into the middle of
    ↪  the list,
    ;; ;; before remembering that a footnote in the text excuses this.
    ;;
    ;; (define (adjoin-term-sparse term term-list)
    ;;   ;; if term is higher order, cons to list
    ;;   ;; if term is lower order than top of list, try to insert in the
    ↪  middle
    ;;   (let ((top-order (order (sparse-first-term term-list))))
    ;;     (cond ((≤ (1+ top-order) (order term))
    ;;             (cons term term-list))
    ;;           (else (adjoin-term-sparse (sparse-first-term term-list)
    ;;                                      (adjoin-term-sparse term
    ;;                                                          (rest-terms
    ↪  term-list)))))))
    ;; (define (adjoin-term-dense term term-list)
    ;;   ;; if term is 1 order above top of list, adjoin
    ;;   ;; if term is more than 1 order above, adjoin 0 and recurse
    ;;   ;; if term is lower order than top of list, try to insert in the
    ↪  middle
    ;;   (let ((top-order (dense-termlist-order term-list)))
    ;;     (cond ((= (1+ top-order) (order term))
    ;;             (cons (coeff term) term-list))
    ;;           ((≤ top-order (order term))
    ;;            (adjoin-term-dense term (cons 0 term-list)))
    ;;           (else (adjoin-term-dense (dense-first-term term-list)
```

```scheme
136    ;;                                      (adjoin-term-dense term
137    ;;                                                          (rest-terms
    ↪  term-list)))))))

139    (define (adjoin-term term term-list)
140      (if (=zero? (coeff term))
141          term-list
142          (if (or (empty-termlist? term-list)
143                  (sparse-termlist? term-list))
144              (adjoin-term-sparse term term-list)
145              (adjoin-term-dense term term-list))))
146    (define (the-empty-termlist) '())
147    (define (rest-terms term-list) (cdr term-list))
148    (define (empty-termlist? term-list) (null? term-list))
149    (define (order term) (first term))
150    (define (coeff term) (second term))
151    (define (add-poly p1 p2)
152      (if (same-variable? (variable p1) (variable p2))
153          (make-poly (variable p1)
154                     (add-terms (terms p1) (terms p2)))
155          (error "Polys not in same var: ADD-POLY" (list p1 p2))))
156    (define (mul-poly p1 p2)
157      (if (same-variable? (variable p1) (variable p2))
158          (make-poly (variable p1)
159                     (mul-terms (terms p1) (terms p2)))
160          (error "Polys not in same var: MUL-POLY" (list p1 p2))))
161    (define (add-terms L1 L2)
162      (cond ((empty-termlist? L1) L2)
163            ((empty-termlist? L2) L1)
164            (else
165             (let ((t1 (first-term L1))
166                   (t2 (first-term L2)))
167               (cond ((> (order t1) (order t2))
168                      (adjoin-term
169                       t1 (add-terms (rest-terms L1) L2)))
170                     ((< (order t1) (order t2))
171                      (adjoin-term
172                       t2 (add-terms L1 (rest-terms L2))))
173                     (else
174                      (adjoin-term
175                       (make-sparse-term (order t1)
176                                         (add (coeff t1) (coeff t2)))
177                       (add-terms (rest-terms L1)
178                                  (rest-terms L2)))))))))
179    (define (mul-terms L1 L2)
180      (if (empty-termlist? L1)
181          (the-empty-termlist)
182          (add-terms (mul-term-by-all-terms (first-term L1) L2)
```

349

```
183                      (mul-terms (rest-terms L1) L2))))

184

185    ;; interface to rest of the system
186    (define (tag p) (attach-tag 'polynomial p))
187    (put 'add '(polynomial polynomial)
188         (lambda (p1 p2) (tag (add-poly p1 p2))))
189    (put 'mul '(polynomial polynomial)
190         (lambda (p1 p2) (tag (mul-poly p1 p2))))
191    (put 'make 'polynomial
192         (lambda (var terms) (tag (make-poly var terms))))
193    (define (if-1st-poly-make-2nd-poly t1 t2)
194      (if (and (eqv? 'polynomial (type-tag t1))
195               (not (eqv? 'polynomial (type-tag t2))))
196          (tag (make-poly (variable (contents t1)) (list t2)))
197          t2))
198    (define (mul-term-by-all-terms t1 L)
199      (if (empty-termlist? L)
200          (the-empty-termlist)
201          (let* ((t2 (first-term L)) ;; If one coeff is a poly, make the
                ↪ other a poly
202                 (t1-coeff (if-1st-poly-make-2nd-poly (coeff t2) (coeff
                        ↪ t1)))
203                 (t2-coeff (if-1st-poly-make-2nd-poly (coeff t1) (coeff
                        ↪ t2))))

204
205            (adjoin-term
206             (make-sparse-term (+ (order t1) (order t2))
207                               (mul t1-coeff t2-coeff))
208             (mul-term-by-all-terms t1 (rest-terms L))))))
209    (define (terms-=zero? L)
210      (or (empty-termlist? L)
211          (and (=zero? (coeff (first-term L)))
212               (terms-=zero? (rest-terms L)))))
213    (define (poly-=zero? p)
214      (terms-=zero? (terms p)))
215    (put '=zero? '(polynomial) poly-=zero?)
216    (define (neg-terms TL)
217      (let ((negterm (if (sparse-termlist? TL)
218                         (lambda(t)
219                           (make-sparse-term (order t) (neg (coeff t))))
220                         (lambda(t)
221                           (neg (coeff t))))))
222        (map negterm TL)))
223    (define (sub-terms T1 T2)
224      (add-terms T1 (neg-terms T2)))
225    (define (neg-poly p)
226      (make-poly (variable p)
227                 (neg-terms (terms p))))
```

```
228      (put 'neg '(polynomial)
229            (lambda (p1) (tag (neg-poly p1))))
230      (define (sub-poly p1 p2)
231        (add-poly p1 (neg-poly p2)))
232      (put 'sub '(polynomial polynomial)
233            (lambda (p1 p2) (tag (sub-poly p1 p2))))
```

```
1    (define (trivial-poly? poly)
2      (and (= 1 (length (terms poly)))
3            (= 0 (order (first-term (terms poly))))))
4    ;;; TODO: get working once cross-variable comparisons can be done
5    (put 'project 'polynomial
6         (lambda (p)
7           (if (not (trivial-poly? p))
8               #f
9               (coeff (first-term (terms p))))))
10   (define (equ?-terms t1 t2)
11     (and (= (order t1)
12             (order t2))
13          (equ? (coeff t1)
14                (coeff t2))))
15   (define (equ?-termlists tl1 tl2)
16           (let iter ((term-list1 tl1)
17                      (term-list2 tl2))
18             (cond ((and (empty-termlist? term-list1)
19                         (empty-termlist? term-list2))
20                    #t)
21                   ((or (empty-termlist? term-list1)
22                        (empty-termlist? term-list2))
23                    #f)
24                   ((equ?-terms (first-term term-list1)
25                                (first-term term-list2))
26                    (iter (rest-terms term-list1)
27                          (rest-terms term-list2)))
28                   (else #f))))
29   (define (equ?-poly p q)
30     (cond ((and (trivial-poly? p)
31                 (trivial-poly? q))
32            (equ? (coeff (first-term (terms p)))
33                  (coeff (first-term (terms q)))))
34           ((same-variable? (variable p) (variable q))
35            (equ?-termlists (terms p) (terms q)))
36           ((trivial-poly? p)
37            (equ?-poly (make-poly (variable q) (terms p))
38                       q))
39           ((trivial-poly? q)
40            (equ?-poly p
41                       (make-poly (variable p) (terms q))))
```

```
42            (else (error "different vars not supported yet : EQU?-POLY" p
              ↪  q))))
43  (put 'equ? '(polynomial polynomial)
44        equ?-poly)
```

```
1  ;; polynomial procedures
2  ;   'done)
3  (define (make-polynomial var terms)
4    ((get 'make 'polynomial) var terms))
```

```
1  (define (add x y) (apply-generic 'add x y))
2  (define (sub x y) (apply-generic 'sub x y))
3  (define (mul x y) (apply-generic 'mul x y))
4  (define (div x y) (apply-generic 'div x y))
5  (define (real-part z) (apply-generic 'real-part z))
6  (define (imag-part z) (apply-generic 'imag-part z))
7  (define (magnitude z) (apply-generic 'magnitude z))
8  (define (angle z) (apply-generic 'angle z))
9  ;; Now we begin to change code for support
10 ;; These will be named slightly different so the primitive
11 ;; functions aren't shadowed
12 (define (square x) (mul x 2))
13 (define (sine x) (apply-generic 'sine x)) ;; (sin)
14 (define (cosine x) (apply-generic 'cosine x)); (cos)
15 (define (square-root x) (apply-generic 'square-root x));(sqrt)
16 (define (arctan x y) (apply-generic 'arctan x y));(atan)
17
18 (define (make-scheme-number n)
19   ((get 'make 'scheme-number) n))
20 (define (install-scheme-number-package)
21   (define (tag x) (attach-tag 'scheme-number x))
22   (put 'add '(scheme-number scheme-number)
23        (lambda (x y) (tag (+ x y))))
24   (put 'sub '(scheme-number scheme-number)
25        (lambda (x y) (tag (- x y))))
26   (put 'mul '(scheme-number scheme-number)
27        (lambda (x y) (tag (* x y))))
28   (put 'div '(scheme-number scheme-number)
29        (lambda (x y) (tag (/ x y))))
30   (put 'sine '(scheme-number)
31        (lambda (x) (tag (sin x))))
32   (put 'cosine '(scheme-number)
33        (lambda (x) (tag (cos x))))
34   (put 'square-root '(scheme-number)
35        (lambda (x) (tag (sqrt x))))
36   (put 'arctan '(scheme-number scheme-number)
37        (lambda (x y) (tag (atan x y))))
```

```scheme
38    (put 'make 'scheme-number (lambda (x) (tag x)))
39    'done)

41  (define (install-rational-package)
42    ;; internal procedures
43    (define (numer x) (first x))
44    (define (denom x) (second x))
45    (define (make-rat n d)
46      (list n d))
47    (define (add-rat x y)
48      (make-rat (add (mul (numer x) (denom y))
49                     (mul (numer y) (denom x)))
50                (mul (denom x) (denom y))))
51    (define (sub-rat x y)
52      (make-rat (sub (mul (numer x) (denom y))
53                     (mul (numer y) (denom x)))
54                (mul (denom x) (denom y))))
55    (define (mul-rat x y)
56      (make-rat (mul (numer x) (numer y))
57                (mul (denom x) (denom y))))
58    (define (div-rat x y)
59      (make-rat (mul (numer x) (denom y))
60                (mul (denom x) (numer y))))
61    ;; interface to rest of the system
62    (define (tag x) (attach-tag 'rational x))
63    (put 'add '(rational rational)
64         (lambda (x y) (tag (add-rat x y))))
65    (put 'sub '(rational rational)
66         (lambda (x y) (tag (sub-rat x y))))
67    (put 'mul '(rational rational)
68         (lambda (x y) (tag (mul-rat x y))))
69    (put 'div '(rational rational)
70         (lambda (x y) (tag (div-rat x y))))
71    (put 'make 'rational
72         (lambda (n d) (tag (make-rat n d))))
73    (put 'numer '(rational) numer)
74    (put 'denom '(rational) denom)
75    (put 'sine '(rational)
76         (lambda(r) ((get 'sine '(scheme-number))
77                     (div (numer r) (denom r)))))
78    (put 'cosine '(rational)
79         (lambda(r) ((get 'cosine '(scheme-number))
80                     (div (numer r) (denom r)))))
81    (put 'square-root '(rational)
82         (lambda(r) ((get 'square-root '(scheme-number))
83                     (div (numer r) (denom r)))))
84    (put 'arctan '(rational)
85         (lambda(r s) ((get 'arctan '(scheme-number))
```

```scheme
86                          (div (numer r) (denom r)) (div (numer s) (denom
                       ↪   s)))))
87     'done)

88

89   ;; for future use
90   (define (numer r)
91     ((get 'numer '(rational))
92      (contents r)))
93   (define (denom r)
94     ((get 'denom '(rational))
95      (contents r)))

96

97   (define (make-rational n d)
98     ((get 'make 'rational) n d))

99

100  (define (install-rectangular-package)
101    ;; internal procedures
102    (define (real-part z) (car z))
103    (define (imag-part z) (cdr z))
104    (define (make-from-real-imag x y) (cons x y))
105    (define (magnitude z)
106      (square-root (add (square (real-part z))
107                        (square (imag-part z)))))
108    (define (angle z)
109      (arctan (imag-part z) (real-part z)))
110    (define (make-from-mag-ang r a)
111      (cons (mul r (cosine a)) (mul r (sine a))))
112    ;; interface to the rest of the system
113    (define (tag x) (attach-tag 'rectangular x))
114    (put 'real-part '(rectangular) real-part)
115    (put 'imag-part '(rectangular) imag-part)
116    (put 'magnitude '(rectangular) magnitude)
117    (put 'angle '(rectangular) angle)
118    (put 'make-from-real-imag 'rectangular
119         (lambda (x y) (tag (make-from-real-imag x y))))
120    (put 'make-from-mag-ang 'rectangular
121         (lambda (r a) (tag (make-from-mag-ang r a))))
122    'done)

123

124  (define (install-polar-package)
125    ;; internal procedures
126    (define (magnitude z) (car z))
127    (define (angle z) (cdr z))
128    (define (make-from-mag-ang r a) (cons r a))
129    (define (real-part z) (mul (magnitude z) (cosine (angle z))))
130    (define (imag-part z) (mul (magnitude z) (sine (angle z))))
131    (define (make-from-real-imag x y)
132      (cons (sqrt (+ (square x) (square y)))
```

354

```
133              (atan y x)))
134       ;; interface to the rest of the system
135       (define (tag x) (attach-tag 'polar x))
136       (put 'real-part '(polar) real-part)
137       (put 'imag-part '(polar) imag-part)
138       (put 'magnitude '(polar) magnitude)
139       (put 'angle '(polar) angle)
140       (put 'make-from-real-imag 'polar
141            (lambda (x y) (tag (make-from-real-imag x y))))
142       (put 'make-from-mag-ang 'polar
143            (lambda (r a) (tag (make-from-mag-ang r a))))
144       'done)
145
146   (define (install-complex-package)
147      (define (make-from-real-imag x y)
148        ((get 'make-from-real-imag 'rectangular) x y))
149      (define (make-from-mag-ang r a)
150        ((get 'make-from-mag-ang 'polar) r a))
151      ;; let's make these generic
152      (define (add-complex z1 z2)
153        (make-from-real-imag (add (real-part z1) (real-part z2))
154                             (add (imag-part z1) (imag-part z2))))
155      (define (sub-complex z1 z2)
156        (make-from-real-imag (sub (real-part z1) (real-part z2))
157                             (sub (imag-part z1) (imag-part z2))))
158      (define (mul-complex z1 z2)
159        (make-from-mag-ang (mul (magnitude z1) (magnitude z2))
160                           (add (angle z1) (angle z2))))
161      (define (div-complex z1 z2)
162        (make-from-mag-ang (div (magnitude z1) (magnitude z2))
163                           (sub (angle z1) (angle z2))))
164      ;; interface to rest of the system
165      (define (tag z) (attach-tag 'complex z))
166      (put 'add '(complex complex)
167           (lambda (z1 z2) (tag (add-complex z1 z2))))
168      (put 'sub '(complex complex)
169           (lambda (z1 z2) (tag (sub-complex z1 z2))))
170      (put 'mul '(complex complex)
171           (lambda (z1 z2) (tag (mul-complex z1 z2))))
172      (put 'div '(complex complex)
173           (lambda (z1 z2) (tag (div-complex z1 z2))))
174      (put 'make-from-real-imag 'complex
175           (lambda (x y) (tag (make-from-real-imag x y))))
176      (put 'make-from-mag-ang 'complex
177           (lambda (r a) (tag (make-from-mag-ang r a))))
178      (put 'real-part '(complex) real-part)
179      (put 'imag-part '(complex) imag-part)
180      (put 'magnitude '(complex) magnitude)
```

```scheme
181      (put 'angle '(complex) angle))

182

183    (define (make-complex-from-real-imag x y)
184      ((get 'make-from-real-imag 'complex) x y))
185    (define (make-complex-from-mag-ang r a)
186      ((get 'make-from-mag-ang 'complex) r a))

187

188    (define (project x)
189      (let ((t1 (type-tag x)))
190        (let ((p (get 'project t1)))
191          (if p
192              (p (contents x))
193              #f))))

194

195    (define (drop x)
196      (if (not (pair? x))
197          x
198          (let ((projected (project x)))
199            (cond ((eqv? projected #f)
200                   x)
201                  ((equ? x projected)
202                   (drop projected))
203                  (else x)))))

204

205    (put 'project 'rational
206         (lambda(num)
207           ((get 'numer '(rational))
208            num)))
209    (put 'project 'real
210         (lambda(num)
211           (round num)))
212    ;; projecting a complex number may return something other than a single
      ↪   number
213    (put 'project 'complex
214         (lambda(num)
215           (let ((r ((get 'real-part '(complex)) num)))
216             (if (number? (contents r))
217                 (make-real (contents r))
218                 r))))

219

220    ;; equality testing with complex numbers also changes
221    (define (equ? x y) (apply-generic 'equ? x y))
222    (define (install-equ?)
223      (put 'equ? '(scheme-number scheme-number)
224           (lambda(x y) (= x y)))
225      (put 'equ? '(complex complex)
226           (lambda(x y)
227             (let ((t1 (type-tag x))
```

```
228                    (t2 (type-tag y)))
229              (cond ((and (eqv? t1 'polar)
230                          (eqv? t2 'polar))
231                     (and (equ? (apply-generic 'magnitude x)
232                                (apply-generic 'magnitude y))
233                          (equ? (apply-generic 'angle x)
234                                (apply-generic 'angle y))))
235                    (else
236                     (and (equ? (apply-generic 'real-part x)
237                                (apply-generic 'real-part y))
238                          (equ? (apply-generic 'imag-part x)
239                                (apply-generic 'imag-part y)))))))))))
240     (put 'equ? '(rational rational)
241          (lambda(x y)
242            (and (equ? ((get 'numer '(rational)) x)
243                       ((get 'numer '(rational)) y))
244                 (equ? ((get 'denom '(rational)) x)
245                       ((get 'denom '(rational)) y)))))
246     (put 'equ? '(real real)
247          (lambda(x y) (= x y))))
```

```
1   (define (=zero? x) (apply-generic '=zero? x))
2   (define (install-=zero?)
3     (put '=zero? '(scheme-number)
4          (lambda(x) (= x 0)))
5     (put '=zero? '(complex)
6          (lambda(x)
7            (apply-generic '=zero? x)))
8     (let ((magnitude (get 'magnitude '(polar))))
9       (put '=zero? '(polar)
10           (lambda(x)
11             (= 0 (magnitude x)))))
12    (let ((real-part (get 'real-part '(rectangular)))
13          (imag-part (get 'imag-part '(rectangular))))
14      (put '=zero? '(rectangular)
15           (lambda(x)
16             (and (=zero? (real-part x))
17                  (=zero? (imag-part x))))))
18    (put '=zero? '(rational)
19         (lambda(x)
20           (or (=zero? ((get 'numer '(rational)) x))
21               (=zero? ((get 'denom '(rational)) x))))))
```

```
1   (load "mattcheck2.scm")
2   <<op-table>>
3   <<type-table>>
4   <<sqrt>>
```

357

```
5    <<echo>>
6    <<smarter-type-tagging>>
7    <<attach-tag-fixed>>
8    <<list-indices>>
9    <<tower-predicates>>
10   <<apply-generic-raiseto>>
11   <<generic-arithmetic-rational-functions>>
12   <<real-package>>
13   <<eqzero?-rf>>
14   <<real-package>>
15
16   <<tower-coercions-rf>>
17
18   <<polynomial-rf-top>>
19   <<polynomial-projection>>
20   <<polynomial-division>>
21   <<polynomial-rf-bottom>>
22
23   <<generic-negation>>
24
25   (install-rational-package)
26   (install-rectangular-package)
27   (install-polar-package)
28   (install-complex-package)
29   (install-scheme-number-package)
30   (install-real-package)
31   (install-equ?)
32   (install-=zero?)
33   (install-polynomial-package)
34   (install-negation)
35
36   <<gen-math-tests>>
37
38   (mattcheck "raise success"
39             (raise 78)
40             (make-rational 78 1))
41   (mattcheck "raise fail"
42             (raise (make-polynomial 'x '((3 2)(1 1))))
43             #f)
44   (mattcheck "apply-generic 1"
45             (add 1 (make-rational 3 5))
46             (make-rational 8 5))
47   (mattcheck "apply-generic 2"
48             (add (make-real 2)
49                  (make-complex-from-mag-ang 3 5))
50             (make-complex-from-real-imag 2.850986556389679
                 ↪  -2.8767728239894153))
51   (mattcheck "drop 1"
```

```
52              (drop (make-complex-from-real-imag 5 0))
53              5)
54  (mattcheck "drop 2"
55              (drop (make-real 5))
56              5)
57  (mattcheck "drop 3"
58              (drop (make-rational 5 1))
59              5)
60  (mattcheck "complex with nested rational"
61              (make-complex-from-real-imag 2 (make-rational 3 5))
62              (cons 'complex (cons 'rectangular (cons 2 (list 'rational 3
        ↪   5)))))
63  (mattcheck "math on complex with nested rational"
64              (add (make-complex-from-real-imag 2 (make-rational 3 5))
65                   (make-complex-from-mag-ang (make-rational 3 5) 2))
66              (cons 'complex (cons 'rectangular
67                                   (cons (list 'rational 8.751559490358574
        ↪   5)
68                                         (list 'rational 28.639461402385223
        ↪   25)))))
69  (mattcheck "polynomial = zero"
70              (=zero? (make-polynomial 'x '((0 0)))))
71  (mattcheck "polynomial ≠ zero" #f
72              (=zero? (make-polynomial 'x '((1 1)(0 1)))))
73
74  (define p1 '((100 1)(2 2)(0 1)))
75  (define p2 (list (list 7 (make-polynomial 'x '((3 2))))))
76  (define p3 '((100 1)(2 1)(0 3)))
77  (define p4 '((100 -1)(2 -2)(0 -1)))
78  (define p5 '(8 2 5 -3 -7))
79  (define p7 '((1 1)(0 1)))
80  (define p6 '((3 1)(0 1)))
81  (define p66 '(1 0 0 1))
82  (define p77 '(1 1))
83  (define p8 (make-polynomial 'x '((2 1) (0 1))))
84  (define p9 (make-polynomial 'x '((3 1) (0 1))))
85  (define p98 (make-rational p9 p8))
86    (mattcheck "polynomial addition"
87              (add (make-polynomial 'x p1)
88                   (make-polynomial 'x p1))
89              '(polynomial x (100 2)(2 4)(0 2)))
90    (mattcheck "polynomial multiplication"
91              (mul (make-polynomial 'x p1)
92                   (make-polynomial 'x p1))
93              '(polynomial x (200 1) (102 4) (100 2) (4 4) (2 4) (0 1)))
94    (mattcheck "nested polynomial addition"
95              (add (make-polynomial 'x p2)
96                   (make-polynomial 'x p1))
```

359

```
97                    '(polynomial x (100 1) (7 (polynomial x (3 2))) (2 2) (0
                  ↪  1)))
98      (mattcheck "nested polynomial mult"
99                    (mul (make-polynomial 'x p2)
100                        (make-polynomial 'x p1))
101                    '(polynomial x (107 (polynomial x (3 2))) (9 (polynomial x
                  ↪  (3 4))) (7 (polynomial x (3 2)))))
102     (mattcheck "negation scheme-number"
103                    (neg 5) -5)
104     (mattcheck "negation rational"
105                    (neg (make-rational 1 5))
106                    (make-rational -1 5))
107     (mattcheck "negation rectangular complex"
108                    (neg (make-complex-from-real-imag 3 7))
109                    (make-complex-from-real-imag -3 -7))
110     (mattcheck "negation polar complex"
111                    (neg (make-complex-from-mag-ang 3 7))
112                    (make-complex-from-mag-ang -3 7))
113
114     (mattcheck "negation polynomials"
115                    (neg (make-polynomial 'x p1))
116                    (make-polynomial 'x p4))
117     (mattcheck "polynomial subtraction"
118                    (sub (make-polynomial 'x p1)
119                        (make-polynomial 'x p3))
120                    '(polynomial x 1 0 -2))
121     (mattcheck "polynomial division 1"
122                    (div (make-polynomial 'x p6)
123                        (make-polynomial 'x p7))
124                    '((polynomial x 1 -1 1) (polynomial x)))
125     (mattcheck "polynomial division 2"
126                    (div (make-polynomial 'x p7)
127                        (make-polynomial 'x p6))
128                    '((polynomial x) (polynomial x 1 1)))
129     (mattcheck "polynomial division 3"
130                    (div (make-polynomial 'x p66)
131                        (make-polynomial 'x p77))
132                    '((polynomial x 1 -1 1) (polynomial x)))
133     (mattcheck "rational functions"
134                    (add p98 p98)
135                    '(rational (polynomial x 2 0 2 2 0 2) (polynomial x 1 0 2 0
                  ↪  1)))
```

```
< ... >
SUCCEED at rational functions
```

360

3.107 Exercise 2.94: GCD of polynomials

3.107.1 Question

Using `div-terms`, implement the procedure `remainder-terms` and use this to
define `gcd-terms` as above. Now write a procedure `gcd-poly` that computes
the polynomial GCD of two polys. (The procedure should signal an error if
the two polys are not in the same variable.) Install in the system a generic
operation `greatest-common-divisor` that reduces to `gcd-poly` for polynomials
and to ordinary `gcd` for ordinary numbers. As a test, try

```
1  (define p1 (make-polynomial
2              'x '((4 1) (3 -1) (2 -2) (1 2))))
3  (define p2 (make-polynomial 'x '((3 1) (1 -1))))
4  (greatest-common-divisor p1 p2)
```

and check your result by hand.

3.107.2 Answer

An important change had to be made here. Dense termlists are padded with
zeroes, while sparse ones are not. This is fine for addition (and subtraction on
the right side) but it can be a problem otherwise. So `dense-first-term` needs to
find the first non-zero coefficient to return, `rest-terms` needs to chop off leading
zeroes for dense lists, and `adjoin-term-sparse` should probably be amended to
skip zeroes so this doesn't happen with sparse lists.

I've also had to rescue some "refugees" from Exercise 2.92, such as `map-terms`.

```
1   (use-srfis '(1))
2   (define (install-polynomial-package) 'no)
3   ;; internal procedures
4   ;; representation of poly
5   (define (make-poly variable term-list)
6     (cons variable
7           (convert-termlist-if-reasonable term-list)))
8   (define (convert-termlist-if-reasonable term-list)
9     (if (empty-termlist? term-list)
10        term-list
11        (cond ((sparse-termlist? term-list)
12               (let ((tl (sort-sparse-terms term-list)))
13                 (if (should-be-sparse? tl)
14                     tl
15                     (sparse-to-dense tl))))
16              ((should-be-sparse? term-list)
17               (dense-to-sparse term-list))
18              (else term-list))))
19  (define (variable p) (car p))
20  (define (terms p) (cdr p))
21  (define (variable? x) (symbol? x))
```

```scheme
22  (define (same-variable? v1 v2)
23    (and (variable? v1) (variable? v2) (eq? v1 v2)))
24
25  ;; representation of terms and term lists
26  (define (sparse-termlist? term-list)
27    (if (empty-termlist? term-list)
28        #t
29        (let ((t (car term-list)))
30          (cond ((and (pair? t) ;; looks like a sparse list
31                      (number? (first t))
32                      (or (pair? (second t)) ;; or sparse with nested poly
33                          (number? (second t)))) #t)
34                ((or (number? t) ;; looks dense
35                     (and (pair? t)
36                          (not (number? (first t))));; or dense with a
                              ↪ nested poly
37                     (eq? #f t)) #f) ;; or dense with an empty entry
38                (else (error "badly formed term: SPARSE-TERMLIST?" t
                      ↪ term-list))))))
39  (define (sort-sparse-terms term-list)
40    (sort term-list (lambda(t1 t2)
41                      (> (order t1)
42                         (order t2)))))
43  (define (should-be-sparse? term-list)
44    ;; sparse or dense term-list -> bool
45    ;; if more than half of term-list is zeroes, return true
46    ;; reasoning:
47    ;; - number of elements used for sparse list is (non-zero elements * 2)
48    ;; - number of elements used for dense list is (highest order)
49    (define (count-zeroes term-list)
50      ;; term-list -> (highest order, zeroes)
51      (let iter ((length 0)
52                 (zeroes 0)
53                 (TL (if (sparse-termlist? term-list)
54                         (sparse-to-dense term-list)
55                         term-list)))
56        (if (empty-termlist? TL)
57            (list zeroes length)
58            (iter (1+ length)
59                  (if (=zero? (car TL))
60                      (1+ zeroes)
61                      zeroes)
62                  (cdr TL)))))
63    (let ((stats (if (sparse-termlist? term-list) ;; do appropriate
            ↪ measurement
64                     (list (- (1+ (order (sparse-first-term term-list)))
65                              (length term-list))
66                           (1+ (order (sparse-first-term term-list))))
```

362

```scheme
                          (count-zeroes term-list))))
      (let ((zero-elements (first stats))
            (highest-order (second stats)))
        (> (/ zero-elements highest-order)
           (/ 1 2)))))
  (define (sparse-to-dense term-list)
    (let iter ((old-list (reverse term-list))
               (new-list (the-empty-termlist)))
      (if (empty-termlist? old-list)
          new-list
          (iter (rest-terms old-list)
                (adjoin-term-dense (sparse-first-term old-list)
                                   new-list)))))
  (define (dense-to-sparse term-list)
    (let iter ((old-list (reverse term-list))
               (old-order 0)
               (new-list (the-empty-termlist)))
      (if (empty-termlist? old-list)
          new-list
          (iter (rest-terms old-list)
                (1+ old-order)
                (adjoin-term-sparse (make-sparse-term old-order
                                                      (coeff (dense-first-⌋
↪    term
↪    old-list)))
                                    new-list)))))
  (define (make-sparse-term order coeff)
    (list order coeff))
  (define (dense-rest-terms term-list)
    (let ((rest (cdr term-list)))
      (if (empty-termlist? rest)
          (the-empty-termlist)
          (let ((c (car rest)))
            (if (eq? #f c)
                (dense-rest-terms rest)
                rest)))))
  (define (rest-terms term-list)
    (if (sparse-termlist? term-list)
        (cdr term-list)
        (dense-rest-terms term-list)))
  (define (dense-termlist-order term-list)
    (1- (length term-list)))
  (define (sparse-termlist-order term-list)
    (order (sparse-first-term term-list)))
  (define (termlist-order term-list)
    (if (sparse-termlist? term-list)
        (sparse-termlist-order term-list)
        (dense-termlist-order term-list)))
```

363

```
113  ;; always O(n)
114  (define (dense-first-term term-list)
115    (let ((c (car term-list)))
116      (if (eq? #f c)
117          (dense-first-term (dense-rest-terms term-list))
118          (make-sparse-term (dense-termlist-order term-list) c))))
119  (define (sparse-first-term term-list)
120    (car term-list))
121  (define (add-sparse-term-to-sparse-term t1 t2)
122    (make-sparse-term (order t1)
123                      (add (coeff t1) (coeff t2))))
124  (define (first-term term-list)
125    (if (sparse-termlist? term-list)
126        (sparse-first-term term-list)
127        (dense-first-term term-list)))
128  (define (adjoin-term-sparse term term-list)
129    ;; if term is higher order, cons to list
130    ;; if equal, replace or add
131    ;; if term is lower order than top of list, try to insert in the middle
132    (if  (empty-termlist? term-list)
133        (cons term term-list)
134        (let ((top-order (order (sparse-first-term term-list))))
135          (cond ((≤ (1+ top-order) (order term))
136                 (cons term term-list))
137                ((= top-order (order term))
138                 (if (=zero? (coeff (sparse-first-term term-list)))
139                     ;; replace if it's zero
140                     (cons term (rest-terms term-list))
141                     ;; and add if it's not
142                     (let ((result (add-sparse-term-to-sparse-term term
143                                                                   (spars↲
                                                                ↪  e-↲
                                                                ↪  fi↲
                                                                ↪  rs↲
                                                                ↪  t-↲
                                                                ↪  te↲
                                                                ↪  rm
                                                                ↪  te↲
                                                                ↪  rm↲
                                                                ↪  -l↲
                                                                ↪  is↲
                                                                ↪  t)↲
                                                                ↪  )))
144                       (cons result
145                             (rest-terms term-list)))))
146                (else (adjoin-term-sparse (sparse-first-term term-list)
147                                          (adjoin-term-sparse term
```

364

```
148                                                          (rest-terms
                                                         ↪  term-li↵
                                                         ↪  st)))))↵
                                                         ↪  )))
149   (define (adjoin-term-dense term term-list)
150     ;; if term is lower order than top of list, try to insert in the middle
151     (let ((top-order (dense-termlist-order term-list)))
152       (cond ((= (1+ top-order) (order term))
153              ;; if term is 1 order above top of list, adjoin
154             (cons (coeff term) term-list))
155             ;; if term shares order with top of list,
156             ((= top-order (order term))
157              (if (=zero? (coeff (dense-first-term term-list)))
158                  ;; replace if it's zero
159                  (cons (coeff term)
160                        (rest-terms term-list))
161                  ;; and add if it's not
162                  (let ((result (add-sparse-term-to-sparse-term term
163                                                                (dense-firs↵
                                                                ↪  t-term
                                                                ↪  term-li↵
                                                                ↪  st))))
164                    (cons (coeff result)
165                          (rest-terms term-list)))))
166             ((< top-order (order term))
167              (adjoin-term-dense term (cons #f term-list)))
168             ;; if term is more than 1 order above, adjoin 0 and recurse
169             (else (adjoin-term-dense (dense-first-term term-list)
170                                      (adjoin-term-dense term
171                                                         (rest-terms term-l↵
                                                         ↪  ist)))))))
172
173   (define (adjoin-term term term-list)
174     (if (=zero? (coeff term))
175         term-list
176         (if (or (empty-termlist? term-list)
177                 (sparse-termlist? term-list))
178             (adjoin-term-sparse term term-list)
179             (adjoin-term-dense term term-list))))
180   (define (the-empty-termlist) '())
181   (define (empty-termlist? term-list)
182     (or (null? term-list)
183         (and (eq? #f (car term-list))
184              (empty-termlist? (cdr term-list)))))
185   (define (order term) (first term))
186   (define (coeff term) (second term))
187   (define (map-terms f tl)
188     ;; f is called (f order coeff)
```

```scheme
189    (let iter ((result '())
190                (term-list tl))
191      (if (empty-termlist? term-list)
192          (reverse result)
193          (iter (cons (let ((t (first-term term-list)))
194                        (f (order t) (coeff t)))
195                      result)
196                (rest-terms term-list)))))
197  (define (map2-terms f tl1 tl2)
198    (let iter ((result '())
199                (term-list1 tl1)
200                (term-list2 tl2))
201      (if (or (empty-termlist? term-list1)
202              (empty-termlist? term-list2))
203          (reverse result)
204          (iter (cons (let ((t1 (first-term term-list))
205                            (t2 (first-term term-list)))
206                        (f (order t1) (coeff t1)
207                           (order t2) (coeff t2)))
208                      result)
209                (rest-terms term-list1)
210                (rest-terms term-list2)))))
211  (define (add-poly p1 p2)
212    (if (same-variable? (variable p1) (variable p2))
213        (make-poly (variable p1)
214                   (add-terms (terms p1) (terms p2)))
215        (error "Polys not in same var: ADD-POLY" (list p1 p2))))
216  (define (mul-poly p1 p2)
217    (if (same-variable? (variable p1) (variable p2))
218        (make-poly (variable p1)
219                   (mul-terms (terms p1) (terms p2)))
220        (error "Polys not in same var: MUL-POLY" (list p1 p2))))
221  (define (add-terms L1 L2)
222    (cond ((empty-termlist? L1) L2)
223          ((empty-termlist? L2) L1)
224          (else
225           (let ((t1 (first-term L1))
226                 (t2 (first-term L2)))
227             (cond ((> (order t1) (order t2))
228                    (adjoin-term
229                     t1 (add-terms (rest-terms L1) L2)))
230                   ((< (order t1) (order t2))
231                    (adjoin-term
232                     t2 (add-terms L1 (rest-terms L2))))
233                   (else
234                    (adjoin-term
235                     (make-sparse-term (order t1)
236                                       (add (coeff t1) (coeff t2)))
```

```scheme
                          (add-terms (rest-terms L1)
                                     (rest-terms L2)))))))))))
(define (mul-terms L1 L2)
  (if (empty-termlist? L1)
      (the-empty-termlist)
      (add-terms (mul-term-by-all-terms (first-term L1) L2)
                 (mul-terms (rest-terms L1) L2))))

;; interface to rest of the system
(define (tag p) (attach-tag 'polynomial p))
(put 'add '(polynomial polynomial)
     (lambda (p1 p2) (tag (add-poly p1 p2))))
(put 'mul '(polynomial polynomial)
     (lambda (p1 p2) (tag (mul-poly p1 p2))))
(put 'make 'polynomial
     (lambda (var terms) (tag (make-poly var terms))))
(define (if-1st-poly-make-2nd-poly t1 t2)
  (if (and (eqv? 'polynomial (type-tag t1))
           (not (eqv? 'polynomial (type-tag t2))))
      (tag (make-poly (variable (contents t1)) (list t2)))
      t2))
(define (mul-term-by-all-terms t1 L)
  (if (empty-termlist? L)
      (the-empty-termlist)
      (let* ((t2 (first-term L)) ;; If one coeff is a poly, make the
              ↪  other a poly
             (t1-coeff (if-1st-poly-make-2nd-poly (coeff t2) (coeff t1)))
             (t2-coeff (if-1st-poly-make-2nd-poly (coeff t1) (coeff
              ↪  t2))))

        (adjoin-term
         (make-sparse-term (+ (order t1) (order t2))
                           (mul t1-coeff t2-coeff))
         (mul-term-by-all-terms t1 (rest-terms L))))))
(define (terms-=zero? L)
  (or (empty-termlist? L)
      (and (=zero? (coeff (first-term L)))
           (terms-=zero? (rest-terms L)))))
(define (poly-=zero? p)
  (terms-=zero? (terms p)))
(put '=zero? '(polynomial) poly-=zero?)
(define (neg-terms TL)
  (let ((negterm (lambda(o c)
                   (make-sparse-term o (neg c)))))
    (map-terms negterm TL)))
(define (sub-terms T1 T2)
  (add-terms T1 (neg-terms T2)))
(define (neg-poly p)
```

```
283      (make-poly (variable p)
284                 (neg-terms (terms p))))
285  (put 'neg '(polynomial)
286       (lambda (p1) (tag (neg-poly p1))))
287  (define (sub-poly p1 p2)
288    (add-poly p1 (neg-poly p2)))
289  (put 'sub '(polynomial polynomial)
290       (lambda (p1 p2) (tag (sub-poly p1 p2))))
```

```
1   (define (remainder-terms tl1 tl2)
2     (second (div-terms tl1 tl2)))
3   (define (gcd-terms a b)
4     (if (empty-termlist? b)
5         a
6         (gcd-terms b (remainder-terms a b))))
7   (define (gcd-poly p q)
8     (cond ((same-variable? (variable p) (variable q))
9            (make-poly (variable p)
10                       (gcd-terms (terms p) (terms q))))
11           ((trivial-poly? p)
12            (gcd-poly (make-poly (variable q) (terms p))
13                      q))
14           ((trivial-poly? q)
15            (gcd-poly p
16                      (make-poly (variable p) (terms q))))
17           (else (error "different vars not supported : GCD-POLY" p q))))
18  (put 'greatest-common-divisor '(polynomial polynomial)
19       (lambda (x y) (tag (gcd-poly x y))))
```

```
1   (define (install-gcd) 'no)
2   (put 'greatest-common-divisor '(scheme-number scheme-number)
3        (lambda (x y) (attach-tag 'scheme-number (gcd x y))))
4   (put 'greatest-common-divisor '(real real)
5        (lambda (x) (attach-tag 'scheme-number (gcd x y))))
6   ;'done)
7   (define (greatest-common-divisor a b)
8     (apply-generic 'greatest-common-divisor a b))
```

```
1   (load "mattcheck2.scm")
2   <<op-table>>
3   <<type-table>>
4   <<sqrt>>
5   <<echo>>
6   <<smarter-type-tagging>>
7   <<attach-tag-fixed>>
8   <<list-indices>>
```

```
 9   <<tower-predicates>>
10   <<apply-generic-raiseto>>
11   <<generic-arithmetic-rational-functions>>
12   <<real-package>>
13   <<eqzero?-rf>>
14   <<real-package>>

15

16   <<tower-coercions-rf>>

17

18   <<polynomial-densepadding-top>>
19   <<polynomial-projection>>
20   <<polynomial-division>>
21   <<polynomial-gcd>>
22   <<polynomial-rf-bottom>>

23

24   <<generic-negation>>
25   <<generic-gcd>>

26

27   (install-rational-package)
28   (install-rectangular-package)
29   (install-polar-package)
30   (install-complex-package)
31   (install-scheme-number-package)
32   (install-real-package)
33   (install-equ?)
34   (install-=zero?)
35   (install-polynomial-package)
36   (install-negation)

37

38   <<gen-math-tests>>

39

40   (mattcheck "raise success"
41             (raise 78)
42             (make-rational 78 1))
43   (mattcheck "raise fail"
44             (raise (make-polynomial 'x '((3 2)(1 1))))
45             #f)
46   (mattcheck "apply-generic 1"
47             (add 1 (make-rational 3 5))
48             (make-rational 8 5))
49   (mattcheck "apply-generic 2"
50             (add (make-real 2)
51                  (make-complex-from-mag-ang 3 5))
52             (make-complex-from-real-imag 2.850986556389679
53             ↪  -2.8767728239894153))
53   (mattcheck "drop 1"
54             (drop (make-complex-from-real-imag 5 0))
55             5)
```

369

```
56  (mattcheck "drop 2"
57             (drop (make-real 5))
58             5)
59  (mattcheck "drop 3"
60             (drop (make-rational 5 1))
61             5)
62  (mattcheck "complex with nested rational"
63             (make-complex-from-real-imag 2 (make-rational 3 5))
64             (cons 'complex (cons 'rectangular (cons 2 (list 'rational 3
       ↪  5)))))
65  (mattcheck "math on complex with nested rational"
66             (add (make-complex-from-real-imag 2 (make-rational 3 5))
67                  (make-complex-from-mag-ang (make-rational 3 5) 2))
68             (cons 'complex (cons 'rectangular
69                                 (cons (list 'rational 8.751559490358574
       ↪  5)
70                                       (list 'rational 28.639461402385223
       ↪  25)))))
71  (mattcheck "polynomial = zero"
72             (=zero? (make-polynomial 'x '((0 0)))))
73  (mattcheck "polynomial ≠ zero" #f
74             (=zero? (make-polynomial 'x '((1 1)(0 1)))))
75
76  (define p1 '((100 1)(2 2)(0 1)))
77  (define p2 (list (list 7 (make-polynomial 'x '((3 2))))))
78  (define p3 '((100 1)(2 1)(0 3)))
79  (define p4 '((100 -1)(2 -2)(0 -1)))
80  (define p5 '(8 2 5 -3 -7))
81  (define p7 '((1 1)(0 1)))
82  (define p6 '((3 1)(0 1)))
83  (define p66 '(1 0 0 1))
84  (define p77 '(1 1))
85  (define p8 (make-polynomial 'x '((2 1) (0 1))))
86  (define p9 (make-polynomial 'x '((3 1) (0 1))))
87  (define p98 (make-rational p9 p8))
88    (mattcheck "polynomial addition"
89             (add (make-polynomial 'x p1)
90                  (make-polynomial 'x p1))
91             '(polynomial x (100 2)(2 4)(0 2)))
92    (mattcheck "polynomial multiplication"
93             (mul (make-polynomial 'x p1)
94                  (make-polynomial 'x p1))
95             '(polynomial x (200 1) (102 4) (100 2) (4 4) (2 4) (0 1)))
96    (mattcheck "nested polynomial addition"
97             (add (make-polynomial 'x p2)
98                  (make-polynomial 'x p1))
99             '(polynomial x (100 1) (7 (polynomial x (3 2))) (2 2) (0
       ↪  1)))
```

```
100    (mattcheck "nested polynomial mult"
101               (mul (make-polynomial 'x p2)
102                    (make-polynomial 'x p1))
103               '(polynomial x (107 (polynomial x (3 2))) (9 (polynomial x
                   ↪  (3 4))) (7 (polynomial x (3 2)))))
104  (define p10 (make-polynomial
105               'x '((4 1) (3 -1) (2 -2) (1 2))))
106  (define p11 (make-polynomial 'x '((3 1) (1 -1))))
107    (mattcheck "negation scheme-number"
108               (neg 5) -5)
109    (mattcheck "negation rational"
110               (neg (make-rational 1 5))
111               (make-rational -1 5))
112    (mattcheck "negation rectangular complex"
113               (neg (make-complex-from-real-imag 3 7))
114               (make-complex-from-real-imag -3 -7))
115    (mattcheck "negation polar complex"
116               (neg (make-complex-from-mag-ang 3 7))
117               (make-complex-from-mag-ang -3 7))
118
119    (mattcheck "negation polynomials"
120               (neg (make-polynomial 'x p1))
121               (make-polynomial 'x p4))
122    (mattcheck "polynomial subtraction"
123               (sub (make-polynomial 'x p1)
124                    (make-polynomial 'x p3))
125               '(polynomial x 1 #f -2))
126    (mattcheck "polynomial division 1"
127               (div (make-polynomial 'x p6)
128                    (make-polynomial 'x p7))
129               '((polynomial x 1 -1 1) (polynomial x)))
130    (mattcheck "polynomial division 2"
131               (div (make-polynomial 'x p7)
132                    (make-polynomial 'x p6))
133               '((polynomial x) (polynomial x 1 1)))
134    (mattcheck "polynomial division 3"
135               (div (make-polynomial 'x p66)
136                    (make-polynomial 'x p77))
137               '((polynomial x 1 -1 1) (polynomial x)))
138    (mattcheck "rational functions"
139               (add p98 p98)
140               '(rational (polynomial x 2 #f 2 2 #f 2) (polynomial x 1 #f
                   ↪  2 #f 1)))
141    (mattcheck "GCD for integers"
142               (greatest-common-divisor 72 30)
143               6)
144    (mattcheck "GCD for polys"
145               (greatest-common-divisor p10 p11)
146               '(polynomial x -1 1 #f))
```

371

```
< ... >
SUCCEED at polynomial subtraction
SUCCEED at polynomial division 1
SUCCEED at polynomial division 2
SUCCEED at polynomial division 3
SUCCEED at rational functions
SUCCEED at GCD for integers
SUCCEED at GCD for polys
```

3.108 Exercise 2.95: a quirk in rational functions

3.108.1 Question

Define P_1, P_2, and P_3 to be the polynomials
$$\begin{array}{ll} P_1: & x^2 - 2x + 1, \\ P_2: & 11x^2 + 7, \\ P_3: & 13x + 5. \end{array}$$

Now define Q_1 to be the product of P_1 and P_2 and Q_2 to be the product of P_1 and P_3, and use `greatest-common-divisor` (@ref{Exercise 2.94}) to compute the GCD of Q_1 and Q_2.

Note that the answer is not the same as P_1. This example introduces non-integer operations into the computation, causing difficulties with the GCD algorithm. [13] To understand what is happening, try tracing `gcd-terms` while computing the GCD or try performing the division by hand.

3.108.2 Answer

```
1   (load "mattcheck2.scm")
2   <<op-table>>
3   <<type-table>>
4   <<sqrt>>
5   <<echo>>
6   <<smarter-type-tagging>>
7   <<attach-tag-fixed>>
8   <<list-indices>>
9   <<tower-predicates>>
10  <<apply-generic-raiseto>>
11  <<generic-arithmetic-rational-functions>>
12  <<real-package>>
13  <<eqzero?-rf>>
14  <<real-package>>
15
16  <<tower-coercions-rf>>
17
```

---

[13] In an implementation like MIT Scheme, this produces a polynomial that is indeed a divisor of Q_1 and Q_2, but with rational coefficients. In many other Scheme systems, in which division of integers can produce limited-precision decimal numbers, we may fail to get a valid divisor.

```
18   <<polynomial-densepadding-top>>
19   <<polynomial-projection>>
20   <<polynomial-division>>
21   <<polynomial-gcd>>
22   <<polynomial-rf-bottom>>
23
24   <<generic-negation>>
25   <<generic-gcd>>
26
27   (install-rational-package)
28   (install-rectangular-package)
29   (install-polar-package)
30   (install-complex-package)
31   (install-scheme-number-package)
32   (install-real-package)
33   (install-equ?)
34   (install-=zero?)
35   (install-polynomial-package)
36   (install-negation)
37
38   (define p1 (make-polynomial 'x '(1 2 1)))
39   (define p2 (make-polynomial 'x '((2 11)(0 7))))
40   (define p3 (make-polynomial 'x '(13 5)))
41   (define q1 (mul p1 p2))
42   (define q2 (mul p1 p3))
43   (echo "P1: " p1)
44   (echo "P2: " p2)
45   (echo "P3: " p3)
46   (echo "Q1: " q1)
47   (echo "Q2: " q2)
48   (echo "GCD(Q1, Q2): " (greatest-common-divisor q1 q2))
```

```
P1:  (polynomial x 1 2 1)
P2:  (polynomial x 11 #f 7)
P3:  (polynomial x 13 5)
Q1:  (polynomial x 11 22 18 14 7)
Q2:  (polynomial x 13 31 23 5)
GCD(Q1, Q2):  (polynomial x 1458/169 2916/169 1458/169)
```

3.109   Exercise 2.96: pseudodivision of polynomials

3.109.1   Text formulas

More precisely, if P and Q are polynomials, let O_1 be the order of P (i.e., the order of the largest term of P) and let O_2 be the order of Q. Let c be the leading coefficient of Q. Then it can be shown that, if we multiply P by the **integerizing factor** $c^{1+O_1-O_2}$, the resulting polynomial can be divided by Q by using the `div-terms` algorithm without introducing any fractions. The operation of multiplying the dividend by this constant and then dividing is sometimes

373

called the **pseudodivision** of P by Q. The remainder of the division is called the **pseudoremainder**.

3.109.2 Questions 1 and 2

1. Implement the procedure `pseudoremainder-terms`, which is just like `re⌋ mainder-terms` except that it multiplies the dividend by the integerizing factor described above before calling `div-terms`. Modify `gcd-terms` to use `pseudoremainder-terms`, and verify that `greatest-common-divisor` now produces an answer with integer coefficients on the example in Exercise 2.95.

2. The GCD now has integer coefficients, but they are larger than those of P_1. Modify `gcd-terms` so that it removes common factors from the coefficients of the answer by dividing all the coefficients by their (integer) greatest common divisor.

3.109.3 Answers 1 and 2

```
1   (define (remainder-terms tl1 tl2)
2     (second (div-terms tl1 tl2)))
3   (define (find-integerizing-factor p q)
4     (let ((o1 (termlist-order p))
5           (o2 (termlist-order q))
6           (c (coeff (first-term q))))
7       (expt c (+ 1 (- o1 o2)))))
8   (define (pseudoremainder-terms tl1 tl2)
9     (let* ((intize (find-integerizing-factor tl1 tl2))
10           (tl1-prime (mul-terms tl1
11                                 (list (make-sparse-term 0 intize)))))
12       (second (div-terms tl1-prime tl2))))
13   (define (gcd-terms a b)
14     (let ((result (if (empty-termlist? b)
15                       a
16                       (gcd-terms b (pseudoremainder-terms a b)))))
17       (if (empty-termlist? result)
18           (the-empty-termlist)
19           (let ((list-gcd (reduce gcd "ERROR"
20                                   (map-terms (lambda(o c) c) result))))
21             (map-terms (lambda(o c)
22                          (make-sparse-term o
23                                            (div c list-gcd)))
24                        result)))))
25   (define (gcd-poly p q)
26     (cond ((same-variable? (variable p) (variable q))
27            (make-poly (variable p)
28                       (gcd-terms (terms p) (terms q))))
29           ((trivial-poly? p)
```

374

```
30              (gcd-poly (make-poly (variable q) (terms p))
31                        q))
32             ((trivial-poly? q)
33              (gcd-poly p
34                        (make-poly (variable p) (terms q))))
35             (else (error "different vars not supported : GCD-POLY" p q))))
36   (put 'greatest-common-divisor '(polynomial polynomial)
37        (lambda (x y) (tag (gcd-poly x y))))
```

```
1    (load "mattcheck2.scm")
2    <<op-table>>
3    <<type-table>>
4    <<sqrt>>
5    <<echo>>
6    <<smarter-type-tagging>>
7    <<attach-tag-fixed>>
8    <<list-indices>>
9    <<tower-predicates>>
10   <<apply-generic-raiseto>>
11   <<generic-arithmetic-rational-functions>>
12   <<real-package>>
13   <<eqzero?-rf>>
14   <<real-package>>
15
16   <<tower-coercions-rf>>
17
18   <<polynomial-densepadding-top>>
19   <<polynomial-projection>>
20   <<polynomial-division>>
21   <<polynomial-gcd-pseudo>>
22   <<polynomial-rf-bottom>>
23
24   <<generic-negation>>
25   <<generic-gcd>>
26
27   (install-rational-package)
28   (install-rectangular-package)
29   (install-polar-package)
30   (install-complex-package)
31   (install-scheme-number-package)
32   (install-real-package)
33   (install-equ?)
34   (install-=zero?)
35   (install-polynomial-package)
36   (install-negation)
37
38   <<gen-math-tests>>
39
```

```
40  (mattcheck "raise success"
41            (raise 78)
42            (make-rational 78 1))
43  (mattcheck "raise fail"
44            (raise (make-polynomial 'x '((3 2)(1 1))))
45            #f)
46  (mattcheck "apply-generic 1"
47            (add 1 (make-rational 3 5))
48            (make-rational 8 5))
49  (mattcheck "apply-generic 2"
50            (add (make-real 2)
51                 (make-complex-from-mag-ang 3 5))
52            (make-complex-from-real-imag 2.850986556389679
            ↪  -2.8767728239894153))
53  (mattcheck "drop 1"
54            (drop (make-complex-from-real-imag 5 0))
55            5)
56  (mattcheck "drop 2"
57            (drop (make-real 5))
58            5)
59  (mattcheck "drop 3"
60            (drop (make-rational 5 1))
61            5)
62  (mattcheck "complex with nested rational"
63            (make-complex-from-real-imag 2 (make-rational 3 5))
64            (cons 'complex (cons 'rectangular (cons 2 (list 'rational 3
            ↪  5)))))
65  (mattcheck "math on complex with nested rational"
66            (add (make-complex-from-real-imag 2 (make-rational 3 5))
67                 (make-complex-from-mag-ang (make-rational 3 5) 2))
68            (cons 'complex (cons 'rectangular
69                                (cons (list 'rational 8.751559490358574
                                ↪  5)
70                                      (list 'rational 28.639461402385223
                                      ↪  25)))))
71  (mattcheck "polynomial = zero"
72            (=zero? (make-polynomial 'x '((0 0)))))
73  (mattcheck "polynomial ≠ zero" #f
74            (=zero? (make-polynomial 'x '((1 1)(0 1)))))
75
76  (define p1 '((100 1)(2 2)(0 1)))
77  (define p2 (list (list 7 (make-polynomial 'x '((3 2))))))
78  (define p3 '((100 1)(2 1)(0 3)))
79  (define p4 '((100 -1)(2 -2)(0 -1)))
80  (define p5 '(8 2 5 -3 -7))
81  (define p7 '((1 1)(0 1)))
82  (define p6 '((3 1)(0 1)))
83  (define p66 '(1 0 0 1))
```

```scheme
84   (define p77 '(1 1))
85   (define p8 (make-polynomial 'x '((2 1) (0 1))))
86   (define p9 (make-polynomial 'x '((3 1) (0 1))))
87   (define p98 (make-rational p9 p8))
88   (mattcheck "polynomial addition"
89             (add (make-polynomial 'x p1)
90                  (make-polynomial 'x p1))
91             '(polynomial x (100 2)(2 4)(0 2)))
92   (mattcheck "polynomial multiplication"
93             (mul (make-polynomial 'x p1)
94                  (make-polynomial 'x p1))
95             '(polynomial x (200 1) (102 4) (100 2) (4 4) (2 4) (0 1)))
96   (mattcheck "nested polynomial addition"
97             (add (make-polynomial 'x p2)
98                  (make-polynomial 'x p1))
99             '(polynomial x (100 1) (7 (polynomial x (3 2))) (2 2) (0 1)))
100  (mattcheck "nested polynomial mult"
101            (mul (make-polynomial 'x p2)
102                 (make-polynomial 'x p1))
103            '(polynomial x (107 (polynomial x (3 2))) (9 (polynomial x (3
         ↪  4))) (7 (polynomial x (3 2)))))
104  (define p10 (make-polynomial
105                'x '((4 1) (3 -1) (2 -2) (1 2))))
106  (define p11 (make-polynomial 'x '((3 1) (1 -1))))
107  (mattcheck "negation scheme-number"
108            (neg 5) -5)
109  (mattcheck "negation rational"
110            (neg (make-rational 1 5))
111            (make-rational -1 5))
112  (mattcheck "negation rectangular complex"
113            (neg (make-complex-from-real-imag 3 7))
114            (make-complex-from-real-imag -3 -7))
115  (mattcheck "negation polar complex"
116            (neg (make-complex-from-mag-ang 3 7))
117            (make-complex-from-mag-ang -3 7))
118
119  (mattcheck "negation polynomials"
120            (neg (make-polynomial 'x p1))
121            (make-polynomial 'x p4))
122  (mattcheck "polynomial subtraction"
123            (sub (make-polynomial 'x p1)
124                 (make-polynomial 'x p3))
125            '(polynomial x 1 #f -2))
126  (mattcheck "polynomial division 1"
127            (div (make-polynomial 'x p6)
128                 (make-polynomial 'x p7))
129            '((polynomial x 1 -1 1) (polynomial x)))
130  (mattcheck "polynomial division 2"
```

377

```
131              (div (make-polynomial 'x p7)
132                   (make-polynomial 'x p6))
133              '((polynomial x) (polynomial x 1 1)))
134    (mattcheck "polynomial division 3"
135              (div (make-polynomial 'x p66)
136                   (make-polynomial 'x p77))
137              '((polynomial x 1 -1 1) (polynomial x)))
138    (mattcheck "rational functions"
139              (add p98 p98)
140              '(rational (polynomial x 2 #f 2 2 #f 2) (polynomial x 1 #f 2
                 ↪ #f 1)))
141    (mattcheck "GCD for integers"
142              (greatest-common-divisor 72 30)
143              6)
144    (mattcheck "GCD for polys"
145              (greatest-common-divisor p10 p11)
146              '(polynomial x -1 1 #f))
147    (let* ((p1p (make-polynomial 'x '(1 2 1)))
148          (p2p (make-polynomial 'x '((2 11)(0 7))))
149          (p3p (make-polynomial 'x '(13 5)))
150          (q1p (mul p1p p2p))
151          (q2p (mul p1p p3p)))
152      (mattcheck "greatest-common-divisor integerizing for polynomials"
153                (greatest-common-divisor q1p q2p)
154                p1p))
```

```
< ... >
SUCCEED at greatest-common-divisor integerizing for polynomials
```

3.110   Exercise 2.97: reducing terms of rational functions

3.110.1   Questions 1 and 2

1. Implement this algorithm as a procedure `reduce-terms` that takes two
   term lists `n` and `d` as arguments and returns a list `nn`, `dd`, which are `n` and
   `d` reduced to lowest terms via the algorithm given above. Also write a
   procedure `reduce-poly`, analogous to `add-poly`, that checks to see if the
   two polys have the same variable. If so, `reduce-poly` strips off the variable
   and passes the problem to `reduce-terms`, then reattaches the variable to
   the two term lists supplied by `reduce-terms`.

2. Define a procedure analogous to `reduce-terms` that does what the original
   `make-rat` did for integers:

```
1      (define (reduce-integers n d)
2        (let ((g (gcd n d)))
3          (list (/ n g) (/ d g))))
```

378

and define `reduce` as a generic operation that calls `apply-generic` to dispatch to either `reduce-poly` (for `polynomial` arguments) or `reduce-integ⌟ers` (for `scheme-number` arguments). You can now easily make the rational-arithmetic package reduce fractions to lowest terms by having `make-rat` call `reduce` before combining the given numerator and denominator to form a rational number. The system now handles rational expressions in either integers or polynomials. To test your program, try the example at the beginning of this extended exercise:

3.110.2   Answers 1 and 2

```scheme
(define (remainder-terms tl1 tl2)
  (second (div-terms tl1 tl2)))
(define (find-integerizing-factor o1 o2 c)
  (expt c (+ 1 (- o1 o2))))
(define (pseudoremainder-terms tl1 tl2)
  (let* ((intize (find-integerizing-factor (termlist-order tl1)
                                           (termlist-order tl2)
                                           (coeff (first-term tl2))))
         (tl1-prime (mul-terms tl1
                               (list (make-sparse-term 0 intize)))))
    (second (div-terms tl1-prime tl2))))

(define (gcd-of-coeffs-in-terms tl)
  (reduce gcd 'ERROR
          (map-terms (lambda(o c) c) tl)))
(define (map-coeffs f tl)
  (map-terms (lambda(o c)
               (make-sparse-term o
                                 (f c)))
             tl))
(define (div-coeffs-by-gcd tl)
  (let ((g (gcd-of-coeffs-in-terms tl)))
    (map-coeffs (lambda(c)(div c g)) tl)))
(define (gcd-terms a b)
  (let ((result (if (empty-termlist? b)
                    a
                    (gcd-terms b (pseudoremainder-terms a b)))))
    (if (empty-termlist? result)
        (the-empty-termlist)
        (div-coeffs-by-gcd result))))
(define (gcd-poly p q)
  (cond ((same-variable? (variable p) (variable q))
         (make-poly (variable p)
                    (gcd-terms (terms p) (terms q))))
        ((trivial-poly? p)
         (gcd-poly (make-poly (variable q) (terms p))
                   q))
```

379

```
38        ((trivial-poly? q)
39         (gcd-poly p
40                   (make-poly (variable p) (terms q))))
41        (else (error "different vars not supported : GCD-POLY" p q))))
42  (put 'greatest-common-divisor '(polynomial polynomial)
43       (lambda (x y) (tag (gcd-poly x y))))
44  (define (reduce-terms n d)
45    (let* ((nd-gcd (gcd-terms n d))
46           (intize (find-integerizing-factor (termlist-order nd-gcd)
47                                             (max (termlist-order n)
48                                                  (termlist-order d))
49                                             (coeff (first-term nd-gcd))))
50           (nn (div-terms (mul-terms
51                           n (list (make-sparse-term
52                                    0 intize)))
53                          nd-gcd))
54           (dd (div-terms (mul-terms
55                           d (list (make-sparse-term
56                                    0 intize)))
57                          nd-gcd)))
58      (if (or (not (empty-termlist? (second nn)))
59              (not (empty-termlist? (second dd))))
60          (error "unexpected remainder from term reduction : REDUCE-TERMS"
61          ↪ n d)
62          (list (div-coeffs-by-gcd (first nn))
                  (div-coeffs-by-gcd (first dd))))))
63  (define (reduce-poly p q)
64    (let ((mp (lambda(target-poly reduced)
65                (let ((v (variable target-poly)))
66                  (list (make-poly v (first reduced))
67                        (make-poly v (second reduced)))))))
68      (cond ((same-variable? (variable p) (variable q))
69             (mp p (reduce-terms (terms p) (terms q))))
70            ((trivial-poly? p)
71             (mp q (reduce-terms (terms p) (terms q))))
72            ((trivial-poly? q)
73             (mp p (reduce-terms (terms p) (terms q))))
74            (else (error "different vars not supported : GCD-POLY" p q)))))
75  (put 'reduce '(polynomial polynomial)
76       (lambda(p q)
77         (let ((result (reduce-poly p q)))
78           (list (tag (first result))
79                 (tag (second result))))))
```

```
1  (define (generic-reduce a b)
2    (apply-generic 'reduce a b))
3  (define (reduce-integers n d)
4    (if (and (integer? n)
```

```
 5            (integer? d))
 6        (let ((g (gcd n d)))
 7          (list (/ n g) (/ d g)))
 8        (list n d)))
 9  (put 'reduce '(scheme-number scheme-number)
10       reduce-integers)
```

```
 1  (define (add x y) (apply-generic 'add x y))
 2  (define (sub x y) (apply-generic 'sub x y))
 3  (define (mul x y) (apply-generic 'mul x y))
 4  (define (div x y) (apply-generic 'div x y))
 5  (define (real-part z) (apply-generic 'real-part z))
 6  (define (imag-part z) (apply-generic 'imag-part z))
 7  (define (magnitude z) (apply-generic 'magnitude z))
 8  (define (angle z) (apply-generic 'angle z))
 9  ;; Now we begin to change code for support
10  ;; These will be named slightly different so the primitive
11  ;; functions aren't shadowed
12  (define (square x) (mul x 2))
13  (define (sine x) (apply-generic 'sine x)) ;; (sin)
14  (define (cosine x) (apply-generic 'cosine x)); (cos)
15  (define (square-root x) (apply-generic 'square-root x));(sqrt)
16  (define (arctan x y) (apply-generic 'arctan x y));(atan)
17
18  (define (make-scheme-number n)
19    ((get 'make 'scheme-number) n))
20  (define (install-scheme-number-package)
21    (define (tag x) (attach-tag 'scheme-number x))
22    (put 'add '(scheme-number scheme-number)
23         (lambda (x y) (tag (+ x y))))
24    (put 'sub '(scheme-number scheme-number)
25         (lambda (x y) (tag (- x y))))
26    (put 'mul '(scheme-number scheme-number)
27         (lambda (x y) (tag (* x y))))
28    (put 'div '(scheme-number scheme-number)
29         (lambda (x y) (tag (/ x y))))
30    (put 'sine '(scheme-number)
31         (lambda (x) (tag (sin x))))
32    (put 'cosine '(scheme-number)
33         (lambda (x) (tag (cos x))))
34    (put 'square-root '(scheme-number)
35         (lambda (x) (tag (sqrt x))))
36    (put 'arctan '(scheme-number scheme-number)
37         (lambda (x y) (tag (atan x y))))
38    (put 'make 'scheme-number (lambda (x) (tag x)))
39    'done)
40
41  (define (install-rational-package)
```

```
42    ;; internal procedures
43    (define (numer x) (first x))
44    (define (denom x) (second x))
45    (define (make-rat n d)
46      (generic-reduce n d))
47    (define (add-rat x y)
48      (make-rat (add (mul (numer x) (denom y))
49                     (mul (numer y) (denom x)))
50                (mul (denom x) (denom y))))
51    (define (sub-rat x y)
52      (make-rat (sub (mul (numer x) (denom y))
53                     (mul (numer y) (denom x)))
54                (mul (denom x) (denom y))))
55    (define (mul-rat x y)
56      (make-rat (mul (numer x) (numer y))
57                (mul (denom x) (denom y))))
58    (define (div-rat x y)
59      (make-rat (mul (numer x) (denom y))
60                (mul (denom x) (numer y))))
61    ;; interface to rest of the system
62    (define (tag x) (attach-tag 'rational x))
63    (put 'add '(rational rational)
64         (lambda (x y) (tag (add-rat x y))))
65    (put 'sub '(rational rational)
66         (lambda (x y) (tag (sub-rat x y))))
67    (put 'mul '(rational rational)
68         (lambda (x y) (tag (mul-rat x y))))
69    (put 'div '(rational rational)
70         (lambda (x y) (tag (div-rat x y))))
71    (put 'make 'rational
72         (lambda (n d) (tag (make-rat n d))))
73    (put 'numer '(rational) numer)
74    (put 'denom '(rational) denom)
75    (put 'sine '(rational)
76         (lambda(r) ((get 'sine '(scheme-number))
77                     (div (numer r) (denom r)))))
78    (put 'cosine '(rational)
79         (lambda(r) ((get 'cosine '(scheme-number))
80                     (div (numer r) (denom r)))))
81    (put 'square-root '(rational)
82         (lambda(r) ((get 'square-root '(scheme-number))
83                     (div (numer r) (denom r)))))
84    (put 'arctan '(rational)
85         (lambda(r s) ((get 'arctan '(scheme-number))
86                       (div (numer r) (denom r)) (div (numer s) (denom
                          ↪  s)))))
87    'done)
88
```

```
89    ;; for future use
90    (define (numer r)
91      ((get 'numer '(rational))
92       (contents r)))
93    (define (denom r)
94      ((get 'denom '(rational))
95       (contents r)))
96
97    (define (make-rational n d)
98      ((get 'make 'rational) n d))
99
100   (define (install-rectangular-package)
101     ;; internal procedures
102     (define (real-part z) (car z))
103     (define (imag-part z) (cdr z))
104     (define (make-from-real-imag x y) (cons x y))
105     (define (magnitude z)
106       (square-root (add (square (real-part z))
107                         (square (imag-part z)))))
108     (define (angle z)
109       (arctan (imag-part z) (real-part z)))
110     (define (make-from-mag-ang r a)
111       (cons (mul r (cosine a)) (mul r (sine a))))
112     ;; interface to the rest of the system
113     (define (tag x) (attach-tag 'rectangular x))
114     (put 'real-part '(rectangular) real-part)
115     (put 'imag-part '(rectangular) imag-part)
116     (put 'magnitude '(rectangular) magnitude)
117     (put 'angle '(rectangular) angle)
118     (put 'make-from-real-imag 'rectangular
119          (lambda (x y) (tag (make-from-real-imag x y))))
120     (put 'make-from-mag-ang 'rectangular
121          (lambda (r a) (tag (make-from-mag-ang r a))))
122     'done)
123
124   (define (install-polar-package)
125     ;; internal procedures
126     (define (magnitude z) (car z))
127     (define (angle z) (cdr z))
128     (define (make-from-mag-ang r a) (cons r a))
129     (define (real-part z) (mul (magnitude z) (cosine (angle z))))
130     (define (imag-part z) (mul (magnitude z) (sine (angle z))))
131     (define (make-from-real-imag x y)
132       (cons (sqrt (+ (square x) (square y)))
133             (atan y x)))
134     ;; interface to the rest of the system
135     (define (tag x) (attach-tag 'polar x))
136     (put 'real-part '(polar) real-part)
```

```scheme
137    (put 'imag-part '(polar) imag-part)
138    (put 'magnitude '(polar) magnitude)
139    (put 'angle '(polar) angle)
140    (put 'make-from-real-imag 'polar
141         (lambda (x y) (tag (make-from-real-imag x y))))
142    (put 'make-from-mag-ang 'polar
143         (lambda (r a) (tag (make-from-mag-ang r a))))
144    'done)

146  (define (install-complex-package)
147    (define (make-from-real-imag x y)
148      ((get 'make-from-real-imag 'rectangular) x y))
149    (define (make-from-mag-ang r a)
150      ((get 'make-from-mag-ang 'polar) r a))
151    ;; let's make these generic
152    (define (add-complex z1 z2)
153      (make-from-real-imag (add (real-part z1) (real-part z2))
154                           (add (imag-part z1) (imag-part z2))))
155    (define (sub-complex z1 z2)
156      (make-from-real-imag (sub (real-part z1) (real-part z2))
157                           (sub (imag-part z1) (imag-part z2))))
158    (define (mul-complex z1 z2)
159      (make-from-mag-ang (mul (magnitude z1) (magnitude z2))
160                         (add (angle z1) (angle z2))))
161    (define (div-complex z1 z2)
162      (make-from-mag-ang (div (magnitude z1) (magnitude z2))
163                         (sub (angle z1) (angle z2))))
164    ;; interface to rest of the system
165    (define (tag z) (attach-tag 'complex z))
166    (put 'add '(complex complex)
167         (lambda (z1 z2) (tag (add-complex z1 z2))))
168    (put 'sub '(complex complex)
169         (lambda (z1 z2) (tag (sub-complex z1 z2))))
170    (put 'mul '(complex complex)
171         (lambda (z1 z2) (tag (mul-complex z1 z2))))
172    (put 'div '(complex complex)
173         (lambda (z1 z2) (tag (div-complex z1 z2))))
174    (put 'make-from-real-imag 'complex
175         (lambda (x y) (tag (make-from-real-imag x y))))
176    (put 'make-from-mag-ang 'complex
177         (lambda (r a) (tag (make-from-mag-ang r a))))
178    (put 'real-part '(complex) real-part)
179    (put 'imag-part '(complex) imag-part)
180    (put 'magnitude '(complex) magnitude)
181    (put 'angle '(complex) angle))

183  (define (make-complex-from-real-imag x y)
184    ((get 'make-from-real-imag 'complex) x y))
```

```scheme
(define (make-complex-from-mag-ang r a)
  ((get 'make-from-mag-ang 'complex) r a))

(define (project x)
  (let ((t1 (type-tag x)))
    (let ((p (get 'project t1)))
      (if p
          (p (contents x))
          #f))))

(define (drop x)
  (if (not (pair? x))
      x
      (let ((projected (project x)))
        (cond ((eqv? projected #f)
               x)
              ((equ? x projected)
               (drop projected))
              (else x)))))

(put 'project 'rational
     (lambda(num)
       ((get 'numer '(rational))
        num)))
(put 'project 'real
     (lambda(num)
       (round num)))
;; projecting a complex number may return something other than a single
↪  number
(put 'project 'complex
     (lambda(num)
       (let ((r ((get 'real-part '(complex)) num)))
         (if (number? (contents r))
             (make-real (contents r))
             r))))

;; equality testing with complex numbers also changes
(define (equ? x y) (apply-generic 'equ? x y))
(define (install-equ?)
  (put 'equ? '(scheme-number scheme-number)
       (lambda(x y) (= x y)))
  (put 'equ? '(complex complex)
       (lambda(x y)
         (let ((t1 (type-tag x))
               (t2 (type-tag y)))
           (cond ((and (eqv? t1 'polar)
                       (eqv? t2 'polar))
                  (and (equ? (apply-generic 'magnitude x)
```

```
232                                      (apply-generic 'magnitude y))
233                          (equ? (apply-generic 'angle x)
234                                (apply-generic 'angle y))))
235                 (else
236                  (and (equ? (apply-generic 'real-part x)
237                             (apply-generic 'real-part y))
238                       (equ? (apply-generic 'imag-part x)
239                             (apply-generic 'imag-part y))))))))))
240     (put 'equ? '(rational rational)
241          (lambda(x y)
242            (and (equ? ((get 'numer '(rational)) x)
243                       ((get 'numer '(rational)) y))
244                 (equ? ((get 'denom '(rational)) x)
245                       ((get 'denom '(rational)) y)))))
246     (put 'equ? '(real real)
247          (lambda(x y) (= x y))))
```

```
1    (load "mattcheck2.scm")
2    <<op-table>>
3    <<type-table>>
4    <<sqrt>>
5    <<echo>>
6    <<smarter-type-tagging>>
7    <<attach-tag-fixed>>
8    <<list-indices>>
9    <<tower-predicates>>
10   <<apply-generic-raiseto>>
11   <<generic-arithmetic-rf-reduce>>
12   <<real-package>>
13   <<eqzero?-rf>>
14   <<real-package>>
15
16   <<tower-coercions-rf>>
17
18   <<polynomial-densepadding-top>>
19   <<polynomial-projection>>
20   <<polynomial-division>>
21   <<polynomial-gcd-reduce>>
22   <<polynomial-rf-bottom>>
23
24   <<generic-negation>>
25   <<generic-gcd>>
26   <<generic-reduce-function>>
27
28   (install-rational-package)
29   (install-rectangular-package)
30   (install-polar-package)
31   (install-complex-package)
```

386

```
32  (install-scheme-number-package)
33  (install-real-package)
34  (install-equ?)
35  (install-=zero?)
36  (install-polynomial-package)
37  (install-negation)
38
39  <<gen-math-tests>>
40
41  (mattcheck "raise success"
42             (raise 78)
43             (make-rational 78 1))
44  (mattcheck "raise fail"
45             (raise (make-polynomial 'x '((3 2)(1 1))))
46             #f)
47  (mattcheck "apply-generic 1"
48             (add 1 (make-rational 3 5))
49             (make-rational 8 5))
50  (mattcheck "apply-generic 2"
51             (add (make-real 2)
52                  (make-complex-from-mag-ang 3 5))
53             (make-complex-from-real-imag 2.850986556389679
                ↪  -2.8767728239894153))
54  (mattcheck "drop 1"
55             (drop (make-complex-from-real-imag 5 0))
56             5)
57  (mattcheck "drop 2"
58             (drop (make-real 5))
59             5)
60  (mattcheck "drop 3"
61             (drop (make-rational 5 1))
62             5)
63  (mattcheck "complex with nested rational"
64             (make-complex-from-real-imag 2 (make-rational 3 5))
65             (cons 'complex (cons 'rectangular (cons 2 (list 'rational 3
                ↪  5)))))
66  (mattcheck "math on complex with nested rational"
67             (add (make-complex-from-real-imag 2 (make-rational 3 5))
68                  (make-complex-from-mag-ang (make-rational 3 5) 2))
69             (cons 'complex (cons 'rectangular
70                                  (cons (list 'rational 8.751559490358574
                                    ↪  5)
71                                        (list 'rational 28.639461402385223
                                          ↪  25)))))
72  (mattcheck "polynomial = zero"
73             (=zero? (make-polynomial 'x '((0 0)))))
74  (mattcheck "polynomial ≠ zero" #f
75             (=zero? (make-polynomial 'x '((1 1)(0 1)))))
```

```scheme
76
77  (define p1 '((100 1)(2 2)(0 1)))
78  (define p2 (list (list 7 (make-polynomial 'x '((3 2))))))
79  (define p3 '((100 1)(2 1)(0 3)))
80  (define p4 '((100 -1)(2 -2)(0 -1)))
81  (define p5 '(8 2 5 -3 -7))
82  (define p7 '((1 1)(0 1)))
83  (define p6 '((3 1)(0 1)))
84  (define p66 '(1 0 0 1))
85  (define p77 '(1 1))
86  (define p8 (make-polynomial 'x '((2 1) (0 1))))
87  (define p9 (make-polynomial 'x '((3 1) (0 1))))
88  (define p98 (make-rational p9 p8))
89  (mattcheck "polynomial addition"
90            (add (make-polynomial 'x p1)
91                 (make-polynomial 'x p1))
92            '(polynomial x (100 2)(2 4)(0 2)))
93  (mattcheck "polynomial multiplication"
94            (mul (make-polynomial 'x p1)
95                 (make-polynomial 'x p1))
96            '(polynomial x (200 1) (102 4) (100 2) (4 4) (2 4) (0 1)))
97  (mattcheck "nested polynomial addition"
98            (add (make-polynomial 'x p2)
99                 (make-polynomial 'x p1))
100           '(polynomial x (100 1) (7 (polynomial x (3 2))) (2 2) (0 1)))
101 (mattcheck "nested polynomial mult"
102           (mul (make-polynomial 'x p2)
103                (make-polynomial 'x p1))
104           '(polynomial x (107 (polynomial x (3 2))) (9 (polynomial x (3
    ↪  4))) (7 (polynomial x (3 2)))))
105 (define p10 (make-polynomial
106                'x '((4 1) (3 -1) (2 -2) (1 2))))
107 (define p11 (make-polynomial 'x '((3 1) (1 -1))))
108 (mattcheck "negation scheme-number"
109           (neg 5) -5)
110 (mattcheck "negation rational"
111           (neg (make-rational 1 5))
112           (make-rational -1 5))
113 (mattcheck "negation rectangular complex"
114           (neg (make-complex-from-real-imag 3 7))
115           (make-complex-from-real-imag -3 -7))
116 (mattcheck "negation polar complex"
117           (neg (make-complex-from-mag-ang 3 7))
118           (make-complex-from-mag-ang -3 7))
119
120 (mattcheck "negation polynomials"
121           (neg (make-polynomial 'x p1))
122           (make-polynomial 'x p4))
```

```
123  (mattcheck "polynomial subtraction"
124             (sub (make-polynomial 'x p1)
125                  (make-polynomial 'x p3))
126             '(polynomial x 1 #f -2))
127  (mattcheck "polynomial division 1"
128             (div (make-polynomial 'x p6)
129                  (make-polynomial 'x p7))
130             '((polynomial x 1 -1 1) (polynomial x)))
131  (mattcheck "polynomial division 2"
132             (div (make-polynomial 'x p7)
133                  (make-polynomial 'x p6))
134             '((polynomial x) (polynomial x 1 1)))
135  (mattcheck "polynomial division 3"
136             (div (make-polynomial 'x p66)
137                  (make-polynomial 'x p77))
138             '((polynomial x 1 -1 1) (polynomial x)))
139  (mattcheck "rational functions (unsimplified, should fail)"
140             (add p98 p98)
141             '(rational (polynomial x 2 #f 2 2 #f 2) (polynomial x 1 #f 2
                ↪ #f 1)))
142  (mattcheck "GCD for integers"
143             (greatest-common-divisor 72 30)
144             6)
145  (mattcheck "GCD for polys"
146             (greatest-common-divisor p10 p11)
147             '(polynomial x -1 1 #f))
148  (let* ((p1p (make-polynomial 'x '(1 2 1)))
149         (p2p (make-polynomial 'x '((2 11)(0 7))))
150         (p3p (make-polynomial 'x '(13 5)))
151         (q1p (mul p1p p2p))
152         (q2p (mul p1p p3p)))
153    (mattcheck "greatest-common-divisor integerizing for polynomials"
154               (greatest-common-divisor q1p q2p)
155               p1p))
156  (mattcheck "rational functions (correctly simplified)"
157             (add p98 p98)
158             '(rational (polynomial x 1 #f #f 1) (polynomial x 1 #f 1)))
```

```
< ... >
FAIL at rational functions (unsimplified, should fail)
expected: (rational (polynomial x 2 #f 2 2 #f 2) (polynomial x 1 #f 2 #f 1))
returned: (rational (polynomial x 1 #f #f 1) (polynomial x 1 #f 1))
< ... >
SUCCEED at greatest-common-divisor integerizing for polynomials
SUCCEED at rational functions (correctly simplified)
```

This exercise has been pretty exhausting. I'll likely take a step back here, maybe work on other projects a bit, and dive back in for Chapter 3.