A Journey Through SICP

Notes, exercises and analyses of Abelson and Sussman

ProducerMatt

March 19, 2023

Contents

1 Introduction Notes

`ID: 5b3d6f68-7dc5-4e82-841a-922a8194cc64`

1.1 Text Foreword

`ID: d7236225-fdc2-4c14-b8cb-773b8d759564`

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

`ID: 0581ae6a-a955-4b91-838f-b133d8fc9da4`

"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

   `ID: ed479e27-c80b-4959-89d9-1587b7966343`

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

`ID: fe3416c7-b95f-41d2-afe3-62c897a454a1`

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

`ID: d38a95d1-239f-4a7b-9aae-0ea369f5db2a`

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

`ID: c0cc921c-fbb2-4a63-a1cc-7c1e477d1e4a`

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)`.

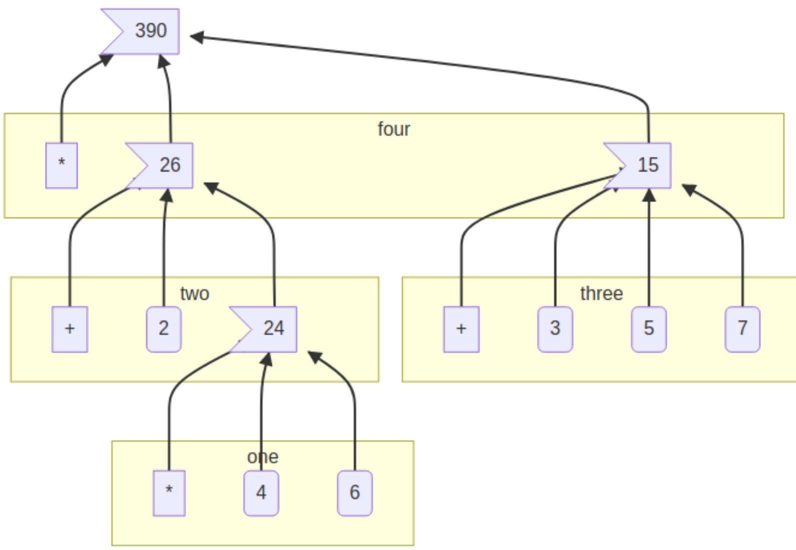
2. It's straightforward to nest combinations in other combinations.

2.3 1.1.3: Evaluating Combinations

`ID: dca55c54-5052-4cfe-b20c-e8cfd379c265`

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

`ID: c5a92594-c625-44d2-a8d0-4b7bf869d862`

We have identified the following in Lisp:

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

Procedure definitions give a name to a compound procedure.

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

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

```scheme
(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

ID: 9f525cef-c10f-4cc0-9476-f94c6b5ff33e

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

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

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

Evaluations can be done in different orders.

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

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

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

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

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

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

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

2.6 1.1.6: Conditional Expressions and Predicates

ID: af3e45b2-e9d8-4c7e-a068-40042f06fb2e

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

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

A shorter form of conditional:

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

If `predicate` is true, `consequent` is returned. Else, `alternative` is returned.

Combining predicates:

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

A small clarification:

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

```
(D) ; => 25 (result of executing procedure D)
```

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

2.7 Exercise 1.1: Trying expressions

ID: dca4ba2c-8ceb-4c65-b8e2-48cc8c9effc8

2.7.1 Question

ID: 135d49f1-3996-4743-a52b-06a022cbd24f

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

ID: 785eed0d-3e37-4972-8108-12dee80e2c8b

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

2.8 Exercise 1.2: Prefix form

ID: 55f52651-703d-4639-a819-80bd992ecd78

2.8.1 Question

ID: b0c2c693-6ddd-40ea-8497-453f4a82d755

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

ID: 928b573d-0aab-46ac-a546-9c7111a6be41

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

1/75

2.9 Exercise 1.3: Conditionals

ID: c90a49f5-d2bc-474b-a304-f6d95a9c3212

2.9.1 Question

ID: ee939cbc-44e8-443e-951b-3059f70cb7f7

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

ID: 86e3bc6c-6fbd-475e-8ba9-47833c38b27d

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

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

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

2.10 Exercise 1.4: Compound expressions

`ID: 1e9dc01f-759b-4522-be54-be32e408da8b`

2.10.1 Question

`ID: 6ba4dc16-42e6-4080-b95a-08d526a82c96`

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:

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

2.10.2 Answer

`ID: 97f32793-fbf2-4443-a2d9-47ab5efef06c`

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:

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

2.11 Exercise 1.5: Applicative vs normal-order evaluation

`ID: ef89d2ff-0e45-4660-8fe8-269c3fff0e96`

2.11.1 Question

`ID: 819c1a91-f83b-4053-88e5-78ab625cdcc0`

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:

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

(define (test x y)
```

```scheme
  (if (= x 0)
      0
      y))
```

Then he evaluates the expression:

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

What behavior will Ben observe with an interpreter that uses applicative-order evaluation? What behavior will he observe with an interpreter that uses normal-order evaluation? Explain your answer. (Assume that the evaluation rule for the special form `if` is the same whether the interpreter is using normal or applicative order: The predicate expression is evaluated first, and the result determines whether to evaluate the consequent or the alternative expression.)

2.11.2 Answer

`ID: 72b34184-067b-430e-ab31-825180d74688`

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

`ID: f3a31fde-4354-4081-a760-94b87d1d41aa`

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

`ID: 4dfd094a-1fb5-4055-b3f6-8d225441e319`

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

ID: 8b07ccff-3309-4b6b-8520-f035fa4432dd

2.14.1 Text code

ID: b5f0b37c-be66-419e-8aa9-caf3cbdeb95a

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

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

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

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

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

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

2.14.2 Question

ID: 27358532-0098-4ded-a4d7-7b99caa3583c

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`:

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

Eva demonstrates the program for Alyssa:

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

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

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

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

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

2.14.3 Answer

`ID: 2c44ee8d-f83f-4661-acfb-fe4aefdf131e`

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

2.15 Exercise 1.7: `sqrt` with small and large numbers

`ID: c4373841-8cfd-4076-86ea-13573c4dd3a6`

2.15.1 Text

`ID: 6e332fab-d501-42c7-84bc-890691a874f3`

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

2.15.2 Question

`ID: 9d4e65f6-d90c-4537-afda-e353aa7c12ef`

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

`ID: f30865d0-5173-4023-b351-d0d717176095`

1. Solving

   `ID: da63b98b-b46c-489f-819d-2968f9bc9b73`

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

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

   This solution can correctly find small and large numbers:

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

```
3162277.6601683795
```

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

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

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

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

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

   `ID: f1a65fe8-2916-47bf-89be-c75ee52e1396`

   (a) Non-optimized

   `ID: 44e1b894-10f0-43b0-b64c-973a6bcfc97f`

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

```
;; 4731.30 <- Benchmark results
```

(b) Optimized

ID: 751f42a4-9c27-4818-b937-5962aaee583f

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

(c) Benchmark results

ID: c434d4dd-1028-4700-9e8b-5eeeaa1af075

Unoptimized	4731.30
Optimized	2518.44

2.15.4 Answer

ID: 202b4dfb-6578-4789-8dbf-b3546e92d8d4

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

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

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

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

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

```
1000000.0
```

So, my definition of `sqrt`:

```
<<average>>
(define (improve guess x)
  (average guess (/ x guess)))
(define (good-enough? guess x)
   (= (improve guess x) guess))
(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x) x)))
(define (sqrt x)
  (sqrt-iter 1.0 x))
```

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

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

2.16 Exercise 1.8: Cube roots

ID: cce3647f-9b12-46b7-a3e6-6ec0b2270926

2.16.1 Question

ID: be02c000-aed9-4970-bc59-e83a7ae76954

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 `square-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

ID: fbaf843c-8346-4874-aa17-4df1034b90d3

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

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

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

7	1.912931182772389
32	3.174802103936399
56	3.825862365544778
100	4.641588833612779

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

2.16.3 Answer

ID: eb13ebca-b39b-4bc9-ba08-9aff7c409bf3

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

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

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

2.17 1.2: Procedures and the Processes They Generate

`ID: 2cc79564-8fc0-4812-a36d-f40f1b832e5e`

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

`ID: 403491c2-9bea-494a-9bdc-404d686ce7c5`

Consider these two procedures for obtaining factorials:

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

```

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

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

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

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

recursive procedure is a procedure whose definition refers to itself.

recursive process is a process that evolves recursively.

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

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

2.19 Exercise 1.9: Peano counting and recursion

`ID: 7bb613b7-a623-42ff-976c-8eff787e6715`

2.19.1 Question

`ID: ca839677-0e8b-4473-8561-e207eab0d5f5`

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.

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

```scheme

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

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

2.19.2 Answer

`ID: c3f9d60e-bdcd-41c2-ab4f-6ca83f0febfb`

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

1. recursive procedure

   `ID: 27210485-cc31-40d1-a1cc-3c40f2ef6136`

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

2. iterative procedure

   `ID: 2246a200-6a61-41f5-803f-480bb8b0cb9c`

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

2.20 Exercise 1.10: Ackermann's Function

`ID: af2058ad-5dbe-413e-9d4e-a249d2b38144`

2.20.1 Question

ID: 1b9d1110-1af6-4b61-9163-b022c50d0c62

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

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

What are the values of the following expressions?

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

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

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

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

2.20.2 Answer

ID: cd179b56-55a7-4736-a088-33fa0e69e82b

1. `f`

   ID: e0e7550f-5456-4f85-9864-a7597599193b

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

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

$$f(n) = 2n$$

2. `g`

   `ID: a780ab66-bc4a-4202-a82c-f8e1e8c03fe3`

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

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

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

3. `h`

   `ID: 6ff1602a-22c8-44c4-8fd1-2356f20b2139`

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

1	2
2	4
3	16
4	65536

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

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

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

2.21 1.2.2: Tree Recursion

`ID: be51794e-b3a2-46aa-ae80-dce13c7ab463`

Consider a recursive procedure for computing Fibonacci numbers:

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))
```

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

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

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

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

2.21.1 Example: Counting change

`ID: 3648fbfb-8c63-41f0-8270-dac38494344d`

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

2.22 Exercise 1.11: More recursion vs iteration

`ID: 5c14394e-0ea2-4493-8022-4aab7d08eca6`

2.22.1 Question

`ID: ea7ae35a-d493-460b-ad6b-85adf126de61`

A function f is defined by the rule that:

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

and

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

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

2.22.2 Answer

`ID: 3856f3ba-fb92-4b5a-b473-ae5f77dee9a8`

1. Recursive

   `ID: 45307adf-a628-466f-8df4-9d3c4c393a4c`

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

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

1	1
3	4
5	25
10	1892

2. Iterative

   `ID: f7c306c6-9592-4083-98a6-8558ea7b3d1c`

   (a) Attempt 1

   `ID: ffe5bb01-bd41-4857-9e1d-48ffa444e353`

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

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

1	1
3	4
5	25
10	1892

It works but it seems wasteful.

(b) Attempt 2

ID: 5ba7b6ff-175e-4658-8878-b6d0fd1bf836

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

```
        (iter 2 1 0 2)))
```

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

1	1
3	4
5	25
10	1892

I like that better.

2.23 Exercise 1.12: Pascal's Triangle

ID: 4f9bba42-afcc-49ba-b8e7-27b3252bf624

2.23.1 Question

ID: d2603acf-c8a3-42c9-89a1-84d27863f691

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

ID: c0e2770f-ab64-4e99-b61b-731f6ec7aab6

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

ID: 26d1bc65-18ae-442f-b2af-43d42ca7dd45

2.24.1 Question

ID: b9ffa2a8-2527-47e4-b08b-6fae0106eeeb

Prove that 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

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

2.24.2 Answer

ID: 60ba8877-b1b5-4aea-8913-e46ccdf76e64

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

1. Fibonacci number generator

   ID: 0a00f599-21ef-4f95-b6ea-c1f4e857556d

```
(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

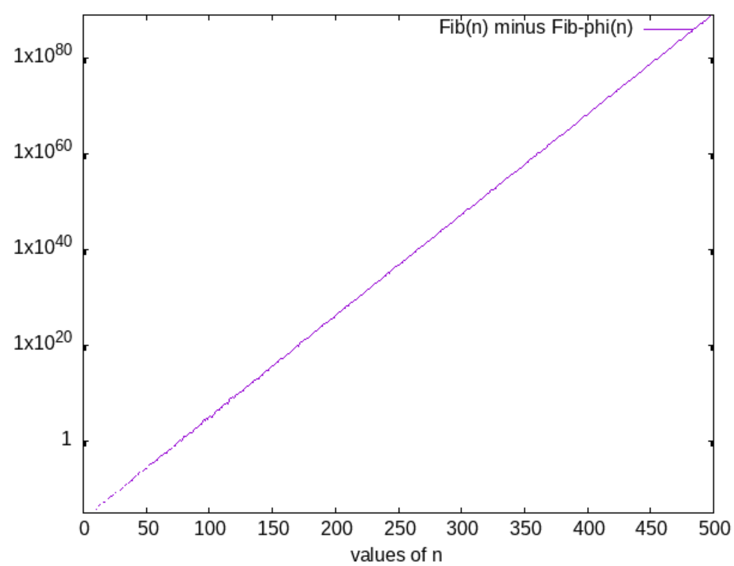
   ID: 4b86e2cf-cc10-4e98-9789-bf744444cc02

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

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

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

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

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



2.25 1.2.3: Orders of Growth

`ID: 788f4aad-14d6-40ae-a32f-acd744b9f700`

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`

`ID: 18db98a1-8e67-4829-939b-44700d20e344`

2.26.1 Text

`ID: 36ba5b4b-ce22-4840-9ecf-dce9c858c4b3`

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

ID: 4f0d0db2-8d42-4ef4-8a96-c6304fc82d12

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

ID: 3b4d300a-acee-4aab-b2de-f40f38948b7a

I want to generate this graph algorithmically.

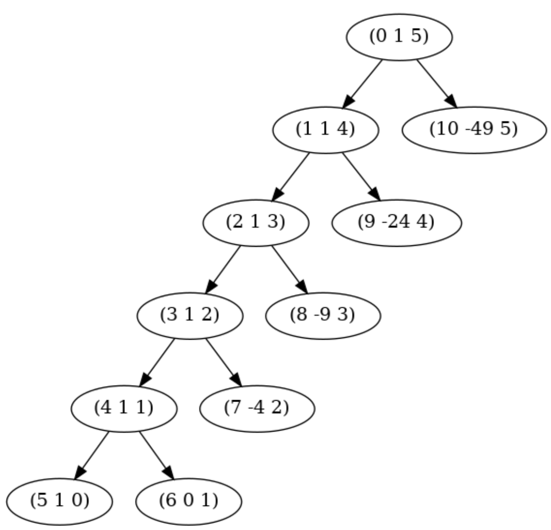
```
;; cursed global
(define bubblecounter 0)
;; Returns # of ways change can be made
;; "Helper" for (cc)
(define (count-change amount)
  (display "digraph {\n") ;; start graph
  (cc amount 5 0)
  (display "}\n") ;; end graph
  (set! bubblecounter 0))

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

(define (first-denomination kinds-of-coins)
  (cond ((= kinds-of-coins 1) 1)
        ((= kinds-of-coins 2) 5)
```

```
        ((= kinds-of-coins 3) 10)
        ((= kinds-of-coins 4) 25)
        ((= kinds-of-coins 5) 50)))
```

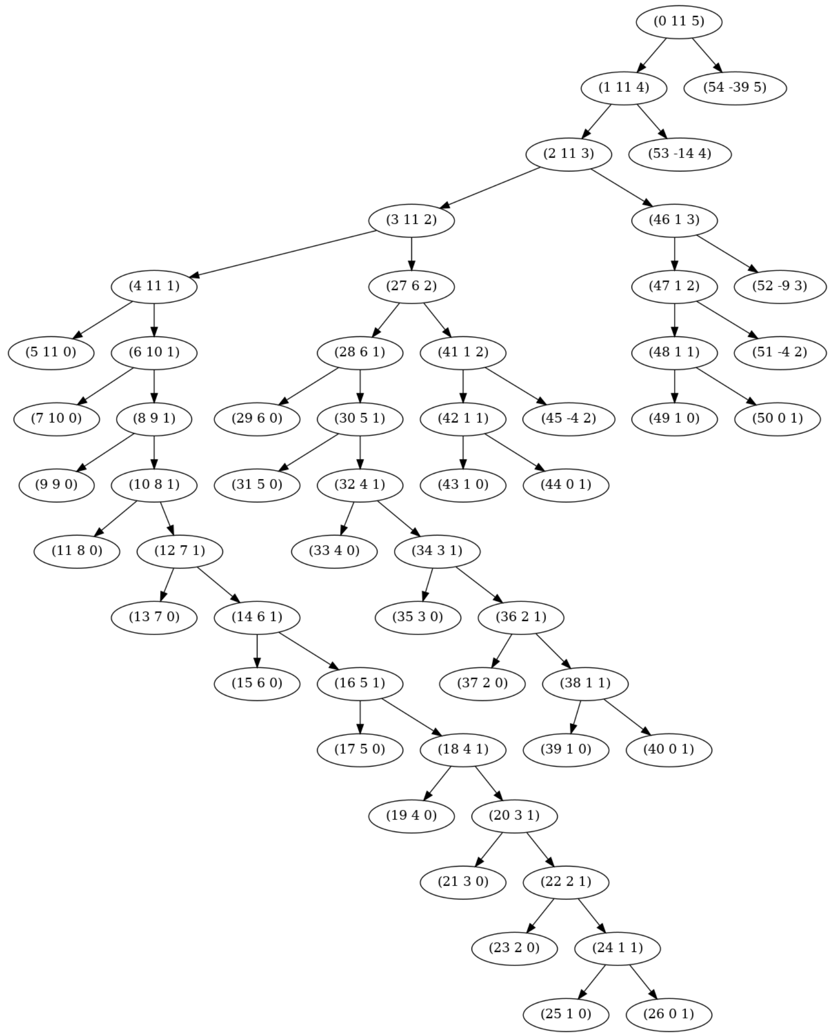
I'm not going to include the full printout of the `(count-change 11)`, here's an example of what this looks like via `1`.

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

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



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



2.26.4 Question B: Analyzing process growth

`ID: 4ec9fbdf-5b5b-46f3-8e62-592695c8df74`

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

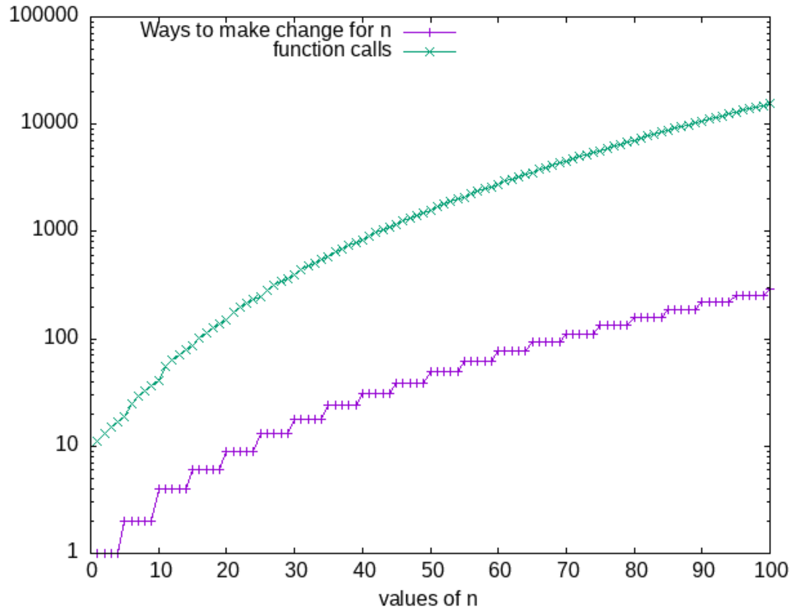
2.26.5 Answer B

`ID: 3c61bcd3-dff2-4fed-a623-f277463c4588`

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

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

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

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

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



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

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

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

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

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

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

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

After seeing a few solutions that I found somewhat confusing, I landed on this awesome article from Codology.net[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

`ID: c409b511-104c-472d-b3ea-645ea1db242b`

2.27.1 Question A

`ID: debe19ef-e946-469a-8549-986110233a82`

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?

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

2.27.2 Answer A

ID: eacd5136-a598-487c-a19a-02dbd45a4032

Let's find out!

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

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

-0.39980345741334	5

p is evaluated 5 times.

2.27.3 Question B

ID: 0c3e1c9a-6435-47bf-b813-c585df53031d

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

ID: f3d8e891-8aa0-4169-89d4-5026cd2fdf7d

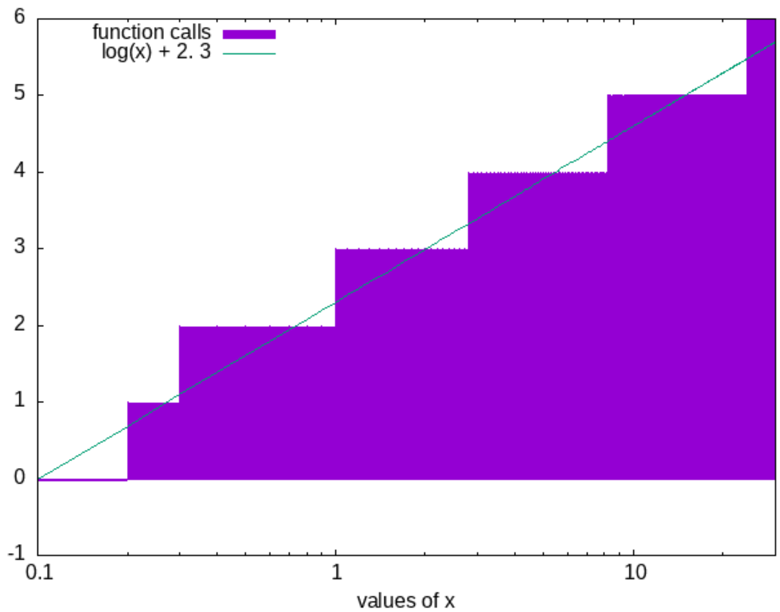
```
(use-srfis '(1))
<<1-15-p-measure>>
(let* ((vals (iota 300 0.1 0.1))
       (sines (map (lambda (i)
                     (cdr (sine i)))
                   vals)))
  (zip vals sines))
```

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

```
    (zip vals sines))
```

Example output:

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



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

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

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

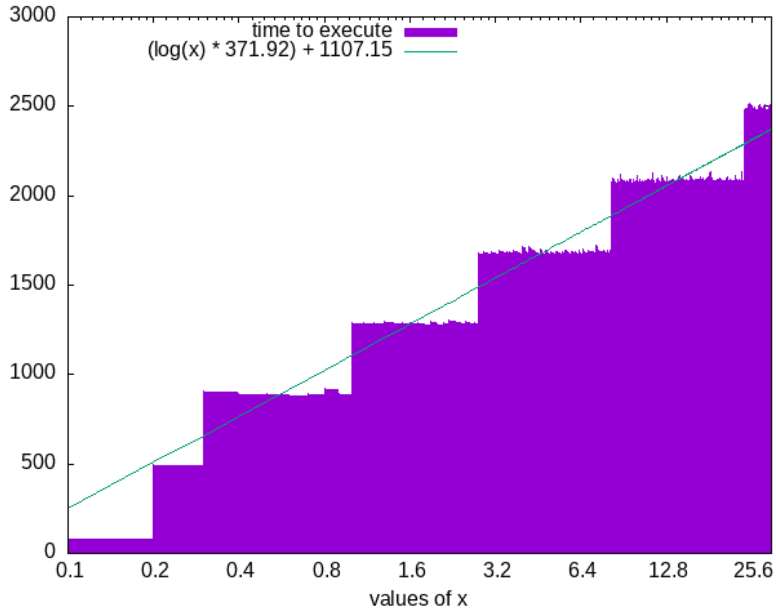
```
(sine 12.15)
(p (sine 4.05))
(p (p (sine 1.35)))
(p (p (p (sine 0.45))))
(p (p (p (p (sine 0.15)))))
(p (p (p (p (p (sine 0.05))))))
(p (p (p (p (p 0.05)))))
(p (p (p (p 0.14950000000000002))))
(p (p (p 0.43513455050000005)))
(p (p 0.9758465331678772))
(p -0.7895631144708228)
-0.39980345741334
```

So I argue the space and time is $\Theta(\log(n))$

We can also prove this for the time by benchmarking the function:

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

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

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

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



1. 1.2.4 Exponentiation

   `ID: 76b874e6-9f56-4167-ba08-573242c66afc`

   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

ID: a98c0259-b567-46bf-b587-12d2e2e4a498

2.28.1 Text

ID: 4b90504b-2032-4417-88fc-49e91ef55ef1

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

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

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

2.28.2 Question

ID: c804b444-1f14-4dd8-8913-b91b0504f32c

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

ID: 59c3a6be-0747-4c53-b4cc-a980cf9c8ef0

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

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

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

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

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

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

2.28.4 Answer

`ID: 0bf21b5e-eba3-4d78-9e73-e4fd5edbf6ca`

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 $a \to a \times b$

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

```
<<fast-expt-iter>>
<<try-these>>
(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)

`ID: 08bb5d53-cd18-4b75-91ca-8561197f87dc`

2.29.1 Question

`ID: c65fcf41-d9f1-46f9-91ef-85e07dd3cf90`

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

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

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

2.29.2 Answer

`ID: 3bc6b3b7-0ee3-43c5-975d-f75061cc764b`

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

Proof it works:

```scheme
<<fast-mult-rec>>
<<try-these>>
(try-these (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)

`ID: 86fcd94b-a1c7-4e54-9bb1-6deb45c950d7`

2.30.1 Question

ID: 67942470-3b36-48f7-a4e5-02b6046edd75

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

ID: 17bfdba8-d5a1-4619-b49f-a2a0d5b1bb57

1. Comparison benchmarks:

   ID: a0b7ff19-6c6a-40f1-87fd-6564d1ecf3ef

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

“fast-mult-rec”	“fast-mult-iter”
196.89	166.35

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

2. Hall of shame

   ID: 90b78921-b998-46d2-8dcd-a2d296a6667b

   Some of my *very* incorrect ideas:

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

   $$ab = \big(a + \Big(\frac{a}{2}\Big)(b-1)\big)$$

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

2.30.3 Answer

ID: 8bd3b472-2415-480c-aad3-4de9011ccaec

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

```scheme
<<fast-mult-iter>>
<<try-these>>
(try-these (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

2.31 Exercise 1.19: Logarithmic fibonacci computing with T

ID: 407ca562-9da1-48a7-b6eb-5d356fd39b7e

2.31.1 Question

ID: dee62f7c-3d69-4510-b1a7-f08b7b8ec7ae

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

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

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

`ID: 2ac06644-ae7a-4025-ac53-436cecff5d7f`

More succinctly put:

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

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

`(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)))))
```

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

```
<<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

ID: bd7688c6-c72b-4739-8109-2332a912a672

```
(define (fib-rec n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib-rec (- n 1))
                 (fib-rec (- n 2))))))
(define (fib n)
  (fib-iter 1 0 0 1 n))

(define (fib-iter a b p q count)
  (cond ((= count 0) b)
        ((even? count)
         (fib-iter a
```

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

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

2.32 1.2.5: Greatest Common Divisor

`ID: 830b140a-2c99-4c28-8197-b5a5d761c624`

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:

$$a\%b = r$$

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

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 k^{th}Fibonacci number.

2.33 Exercise 1.20: Inefficiency of normal-order evaluation

ID: 663150d8-1c0b-468f-8db4-018423b53fc0

2.33.1 Text

ID: ae5a7fe3-5ff5-4a85-a964-10dec6869216

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

2.33.2 Question

ID: 716fdfd6-be01-4639-a23e-e77713a2b719

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

ID: fad2aa9c-9dab-464f-97ab-0f2d21da4608

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

   ID: f613f599-bada-4fe3-ba42-db9641359d4d

   ```
   call (gcd 206 40)
   (if)
   (gcd 40 (remainder 206 40))
   eval remainder before call
   call (gcd 40 6)
   (if)
   (gcd 6 (remainder 40 6))
   ```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
    (gcd (remainder (remainder 40 (remainder 206 40)) (remainder
    ↪  (remainder 206 40) (remainder 40 (remainder 206 40))))
    ↪  (remainder a  (remainder (remainder 40 (remainder 206 40))
    ↪  (remainder (remainder 206 40) (remainder 40 (remainder 206
    ↪  40)))))))

; eval 4 remainders
(remainder (remainder 206 40) (remainder 40 (remainder 206 40)))
; => 2
```

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

2.34 1.2.6: Example: Testing for Primality

ID: e493130a-61e6-4e34-ba26-12317d664402

Two algorithms for testing primality of numbers.

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

2. $\Theta(\log n)$: Given a number n, pick a random number $a < n$ and compute the remainder of a^n modulo n. If the result is not equal to a, then n is certainly not prime. If it is a, then chances are good that n is prime. Now pick another random number a and test it with the same method. If it also satisfies the equation, then we can be even more confident that n is prime. By trying more and more values of a, we can increase our confidence in the result. This algorithm is known as the Fermat test.

   Fermat's Little Theorem: If n is a prime number and a is any positive integer less than n, then a raised to the n^{th} power is congruent to a modulo n. [Two numbers are *congruent modulo* n if they both have the same remainder when divided by n.]

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

2.35 Exercise 1.21

ID: 2da88cf0-4a59-4288-88f3-f9d7394720b6

2.35.1 Text

ID: 919355bd-d6f7-4df8-b80d-f2691ea0b4d8

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

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

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

2.35.2 Question

`ID: 7f5e1e80-6aae-485e-ada9-858fc47dbeae`

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

2.35.3 Answer

`ID: 88b6dc2a-a533-4942-92be-bf16f41cb3f6`

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

199	1999	7

2.36 Exercise 1.22

`ID: f3a9bc52-9915-4be3-9dba-718019f3fa6d`

2.36.1 Question

`ID: 85a19748-38ef-422c-9940-2aada7ed308f`

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

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

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

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

2.36.2 Answer

`ID: e13b44f5-69de-4948-a857-a1c4530ce998`

1. Part 1

   `ID: 0f4b2e49-36bd-466e-9bb7-48f6b327f55f`

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

   ```
   <<timed-prime-test-txt>>
   (define (search-for-primes minimum goal)
     (define m (if (even? minimum)
                   (+ minimum 1)
                   (minimum)))
   ```

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

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

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

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

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

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

2. Part 2

ID: 98cc6660-fe76-4b19-a036-afb4dace8757

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

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

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

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

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

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

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

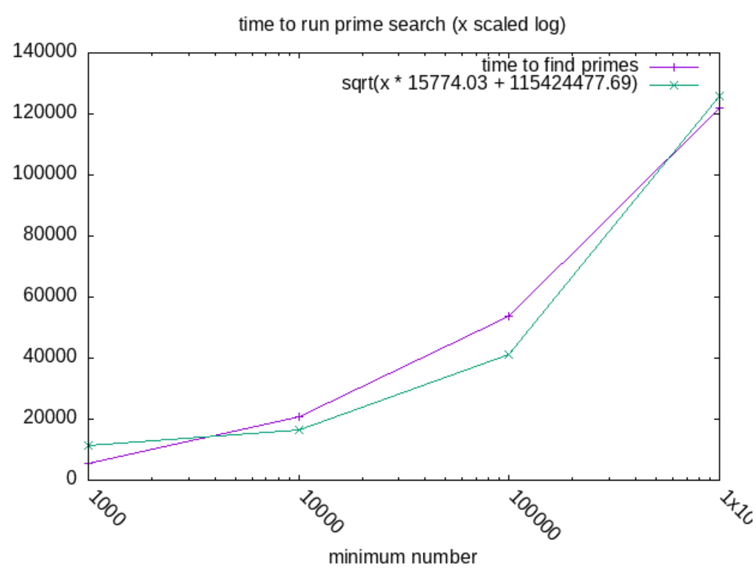
```
(define (search-for-primes minimum goal)
  (define m (if (even? minimum)
                (+ minimum 1)
                (minimum)))
  (search-for-primes-iter m '() goal))
(define (search-for-primes-iter n lst goal)
  (if (= goal 0)
      lst
      (let ((x (prime? n)))
        (if (not (equal? x #f))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

I can benchmark these functions like so:

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

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

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

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

5425.223086	20772.332491	53577.240193	121986.712395



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

2.37 Exercise 1.23

ID: 9e3a40e2-3c5d-45ba-af3c-969651c94297

2.37.1 Question

ID: 6c0c0cec-b4be-4739-8bae-14fe069e22cd

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

2.37.2 A Comedy of Error (just the one)

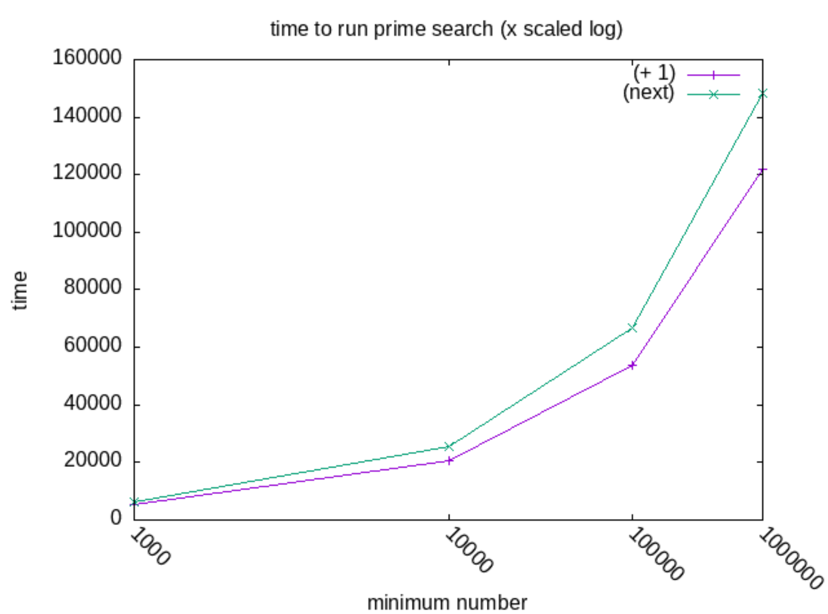
ID: 6d687d6f-1b84-474f-89e8-194462e90b88

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

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

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

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

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

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

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

6456.538118	25550.757304	66746.041644	148505.580638

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



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

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

2.37.3 Answer

`ID: b68121ea-e528-442f-8046-09eee174ef97`

Ok, let's try that again.

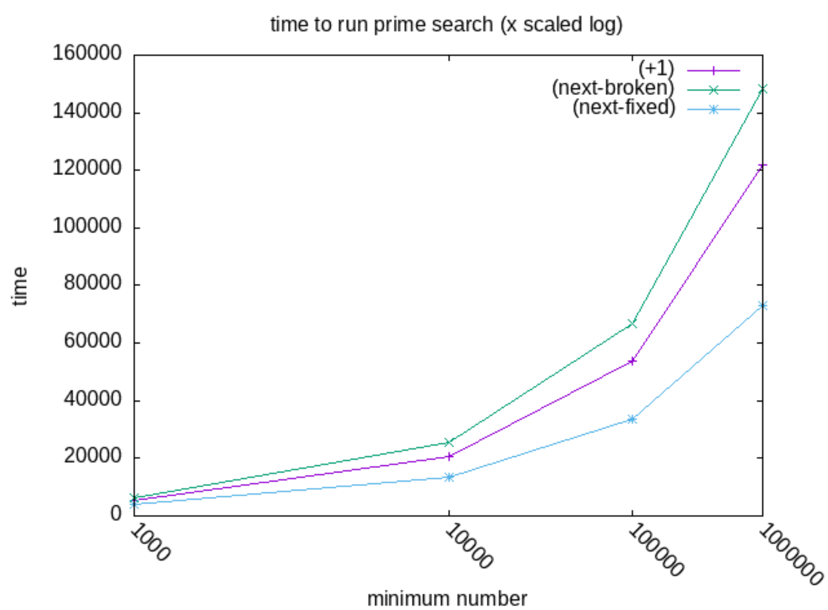
```
<<square>>
(define (smallest-divisor n)
  (find-divisor n 2))

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

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

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

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

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

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

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

3863.7424	13519.209814	33520.676384	73005.539932

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



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

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

Defining a new average that takes arbitrary numbers of arguments:

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

Using it for percentage comparisons:

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

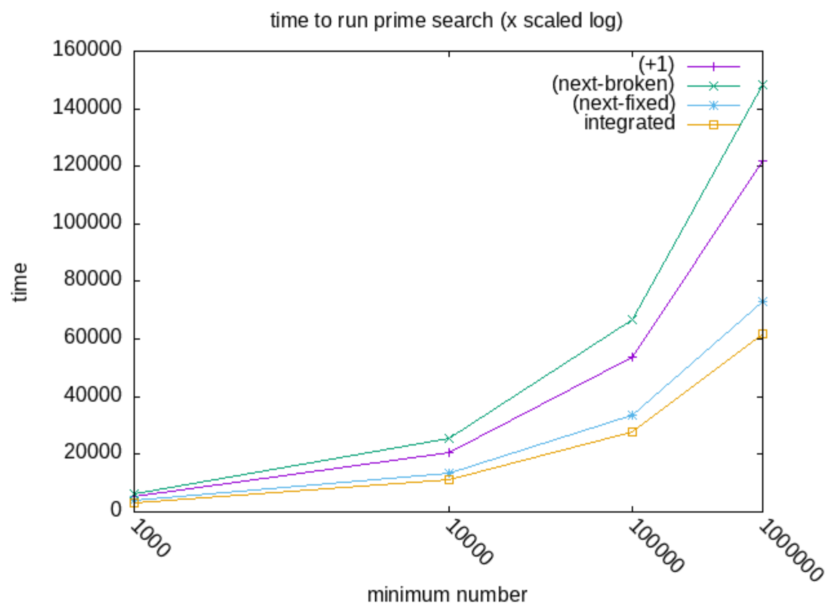
2. Because it only needs to be run once, it doesn't need to be a separate function at all.

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

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

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

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

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

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

3151.259574	11245.20428	27803.067944	61997.275154

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



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

2.38 Exercise 1.24

ID: 24f2cec1-17bd-456c-84e9-fe5c685301d5

2.38.1 Text

ID: 9259fc75-fc98-4f03-b755-6fab60dc38f6

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

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

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

2.38.2 Question

ID: d1464d89-e40f-4220-83a2-07bd83cdba29

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?

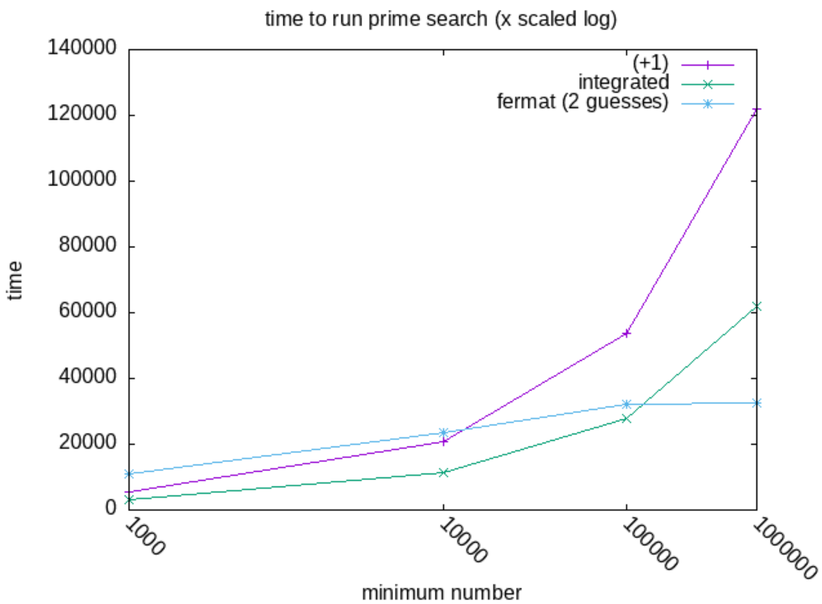
2.38.3 Answer

ID: 32df4fe3-ba2b-4580-b6fc-ac2d683d69a4

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

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

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

11175.799722	23518.62116	32150.745642	32679.766448

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

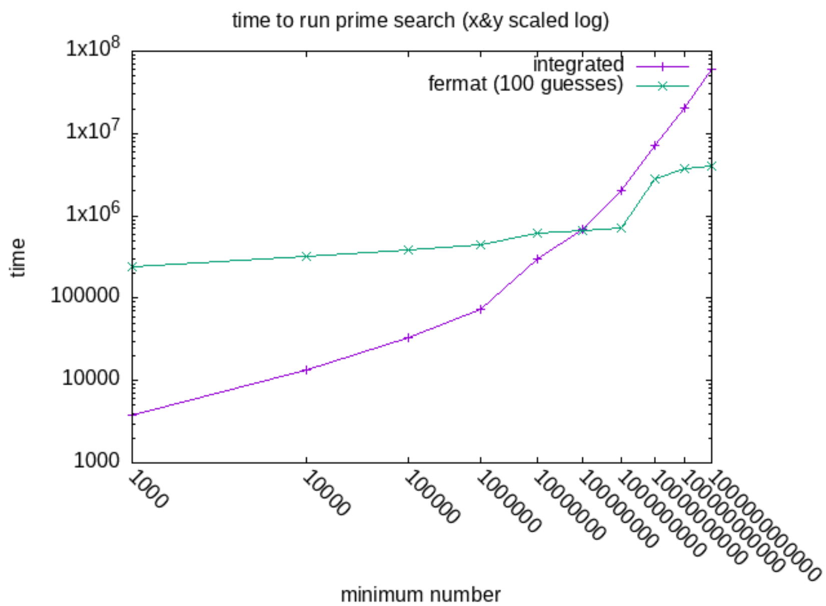


It definitely looks to be advancing much slower than the other methods. I'd like to see more of the function.

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

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

```
        (cadr (mattbench2 (lambda() (f 10000000 3))
        ↪  benchmark-iterations))
        (cadr (mattbench2 (lambda() (f 100000000 3))
        ↪  benchmark-iterations))
        (cadr (mattbench2 (lambda() (f 1000000000 3))
        ↪  benchmark-iterations))
        (cadr (mattbench2 (lambda() (f 10000000000 3))
        ↪  benchmark-iterations))
        (cadr (mattbench2 (lambda() (f 100000000000 3))
        ↪  benchmark-iterations))
        (cadr (mattbench2 (lambda() (f 1000000000000 3))
        ↪  benchmark-iterations))))

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

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

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

(print-row
 (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

`ID: ed832d1f-dde0-45db-8891-014b0c9daff9`

2.39.1 Question

`ID: 76148d28-d72e-4be4-ab2a-d96c2f100dea`

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

```
(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

`ID: f4e6ce08-2adb-4c3f-a565-096aea235bfc`

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

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

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

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

2.40 Exercise 1.26

`ID: 77720381-e1ba-460a-aef5-08292daa3384`

2.40.1 Question

ID: eee5142a-175d-4ae0-b9cb-6c1fd079ba0a

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

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

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

2.40.2 Answer

ID: 9e8c8cdc-7517-4422-939d-f22026d6aa91

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

ID: 6a54e493-0e3d-4bb1-84ff-88196dda3f15

2.41.1 Question

ID: 9efef8fb-5bcf-45bb-96b0-76902a307f51

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   1105   1729   2465   2821   6601

2.41.2 Answer

ID: c7e064e7-5ace-4ef2-9e29-469b2573cb7b

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

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

2.42 Exercise 1.28

ID: 99be304b-3e68-4500-8c6d-f4300fbcdf36

2.42.1 Question

ID: dbc95aaa-41b7-44ff-b563-8d87e448682d

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

ID: 1a682537-2536-4aca-afcf-bd46ea38d862

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

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

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

```
(use-srfis '(1))
<<expmod>>
<<fermat-prime?>>
<<find-divisor-faster-real>>
(define (prime? n)
  (= n (smallest-divisor n)))
(call-with-output-file "Data/carmichael-verification.txt" (lambda(port)
     (write (filter
             (lambda(x) (and (fermat-prime? x)
                             (not (prime? x))))
             (iota (- 1000000 1000) 1000))
            port)))
```

This will be useful in various future functions:

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

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

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

2.42.3 Answer

ID: cd30d1cc-10c2-4e3c-a97e-2154b2ea4bf2

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

```
(define (mr-test n)
  (define (try-it a)
    (let ((it (expmod-mr a n n)))
      (or (= it a)
```

```
           (= it 0))))
    (try-it (+ 1 (random (- n 1)))))
```

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

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

```
<<square>>
(define (expmod-mr base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
         ;; Keep result and remainder seperate
         (let* ((result (expmod-mr base (/ exp 2) m))
                (rem (remainder (square result) m)))
           (if (and (not (= result 1))
                    (not (= result (- m 1)))
                    (= 1 rem))
               0 ;; non-trivial sqrt mod 1 is found
               rem)))
        (else
         (remainder
          (* base (expmod-mr base (- exp 1) m))
          m))))
```

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

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

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

One more time.

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

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

`ID: 27250639-d875-4ff2-9a74-1274c57310e1`

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

2.44 1.3.1: Procedures as Arguments

`ID: 5fa080a9-ceb5-41c8-81b7-d98b449b0287`

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

For the program:

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

Which is equivalent to:

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

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

ID: 22b91a3d-9eb1-4d5f-9f28-0c085c98d58f

2.45.1 Text

ID: c323c9f4-932d-4d36-865e-a18e76f566b4

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

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

2.45.2 Question

ID: 8b97d855-2b4f-4c1c-aec3-815efe988cc3

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

ID: 6d0c9bfa-5b69-4994-b5aa-ed531a2ecae8

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

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

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

(define iterations 100000) ;; benchmark iterations
(define (run-test1)
  (integral cube 0.0 1.0 0.0008))
(define (run-test2)
  (int-simp cube 0.0 1.0 1000.0))
(print-table (list (list "integral dx:0.0008" "int-simp i:1000")
                   (list (run-test1) (run-test2))
                   (list (cadr (mattbench2 run-test1 iterations))
```

```
                    (cadr (mattbench2 run-test2 iterations))))
              #:colnames #t)
```

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

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

2.46 Exercise 1.30

`ID: e54f9c79-9c6e-40e3-8d72-c3b38277249f`

2.46.1 Question

`ID: 0e7c46b4-644b-4448-b4a4-8a93967ac9b2`

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

`ID: a6d60e74-2b88-4332-adf2-e8e0a31b856e`

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

Let's check the stats!

recursive	iterative
30051.080005	19568.685587

2.47 Exercise 1.31

`ID: 3b741046-fc06-4a6f-a4f8-dbd1656de562`

2.47.1 Question A.1

`ID: b2a6bb77-2e48-4c1e-a8bb-3197667412fb`

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

ID: c0d12247-b8ec-4627-beb3-a3ea241588f6

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

2.47.3 Question A.2

ID: 9cdeebe4-d20b-477d-b261-1cf6aebe163c

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

2.47.4 Answer A.2

ID: 1f80f423-6bcf-4ed0-b339-f51dbe117833

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.

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

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

(display (factorial 7))
```

2.47.5 Question A.3

ID: aabc1dbe-2fd6-4569-9dfc-7cbea0f9286f

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

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

2.47.6 Answer A.3

ID: 8dd42248-4caa-4eba-bea6-92130574a000

Once this equation is encoded, you just need to multiply it by two to get π. Fun fact: the formula is slightly wrong, it should start the series with $\frac{1}{2}$.

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

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

3.1415769458228726

2.47.7 Question B

ID: 8e708af0-405b-4500-8c24-a8b2b14add83

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

ID: ecd3adca-6ab8-4a51-82af-8d45f3f84410

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

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

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

(define iterations 50000)
(print-table
 (list (list "iterative" "recursive")
       (list (cadr (mattbench2 (lambda()(pi-product 1000)) iterations))
             (cadr (mattbench2 (lambda()(pi-product-rec 1000))
             ↪  iterations))))
 #:colnames #t)
```

iterative	recursive
1267118.0538	3067085.5323

2.48 Exercise 1.32

ID: f4a4d034-09b6-4ff5-968a-69b74b5ed370

2.48.1 Question A

ID: acee9589-f807-4d91-acf4-f432887f9269

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

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

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

2.48.2 Answer A

ID: b6e058f3-443d-455d-9b2a-42e6a0726c2c

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

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

```
        (iter (next a)
              (combiner result (term a)))))
  (iter a null-value))
```

```
<<accumulate-iter>>

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

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

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

(display (factorial 7))
```

```
5040
```

2.48.3 Question B

ID: 5af0317d-e560-4a81-b2ae-dfdc061356e1

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

ID: 89f3a10e-0d2d-42df-8261-c850d9ca0152

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

2.49 Exercise 1.33

ID: 0f988768-6d07-4947-a89d-bc294a2da075

2.49.1 Question A

ID: cc9235e8-f2a4-4ee0-9ec0-eae8f7c42f19

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

ID: 398f19b8-8908-467d-9b84-b921525bab17

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

2.49.3 Question B

ID: f9acb49a-f3e7-4b81-9a67-fb0c2468d60d

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

1. A

   ID: d52ed48f-0ada-438c-92a4-81a698b86434

   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)

   ```
   (load "mattcheck.scm")
   (define (square x)
     (* x x))
   <<filtered-accumulate-iter>>
   <<expmod-mr2>>
   <<mr-test2>>
   <<mr-prime>>
   (define mr-times 100)
   (define (prime? x)
   ```

```
    (mr-prime? x mr-times))
  (define (prime-sum a b)
    (filtered-accumulate-iter prime? + 0
                              square a 1+ b))

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

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

2. B

`ID: 1c20e096-168c-42d6-a625-14a62757c355`

> 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 $\text{GCD}(i, n) = 1$.

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

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

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

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

```
                    (alternate 100))
```

SUCCEED at Ex_1-33B

2.50 1.3.2: Constructing Procedures Using lambda

ID: bebc1b91-d31b-44dd-84af-04b0da3604ba

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

ID: a8aa7863-ffd5-49b2-9111-0deba26721fb

2.51.1 Question

ID: 9d5702e5-4f3a-4664-8eaa-5fc05ee79095

Suppose we define the procedure

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

Then, we have

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

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

2.51.2 Answer

ID: 572f2214-dd1b-41a3-aab8-1f689eeb18f8

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

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

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

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

2.52 1.3.3 Procedures as General Methods

ID: 0d57f264-3dde-4357-9bcf-8ae957350f5d

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

ID: e31d236d-e1ce-4869-81b0-8b79d34638ca

2.53.1 Text

ID: be2873e9-e336-4dc4-bf2f-bd853e3eaf9e

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

```
(define tolerance 0.00001)

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

```
        (try next))))
  (try first-guess))
```

2.53.2 Question

ID: 34a47af6-9e71-451d-8c6f-0698d43b8e7b

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

2.53.3 Answer

ID: b34d7cd6-a871-4e8c-9988-df80f50af1aa

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

(display golden-ratio)
```

```
1.6180327868852458
```

2.54 Exercise 1.36

ID: 14a2f8ac-4e60-4c39-8df7-b3c4991e03d0

2.54.1 Question

ID: b53a92e6-8307-4e5f-823e-2670eb1e7e96

> 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

ID: 5742a471-6570-4168-8d26-bf7fd9458c84

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

```
(use-modules (ice-9 format))
(define tolerance 0.00001)

(define (fixed-point f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2))
       tolerance))
  (define (try guess)
    (let ((next (f guess)))
      (format #t "~&~a~%" next)
      (if (close-enough? guess next)
          next
          (try next))))
  (try first-guess))
```

```
<<close-enough>>
<<fixed-point-debug>>
(fixed-point (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

ID: ea7098b0-6504-4aaf-89e1-894f16593e0b

2.55.1 Question A

ID: cd886b09-1c12-4a40-966f-685ccf39cd73

An infinite continued fraction is an expression of the form

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

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

`ID: 2ee7e946-ede6-458d-a353-82cc67999c8a`

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

`ID: f58d571e-0643-4f24-8db5-1e1431ee4315`

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

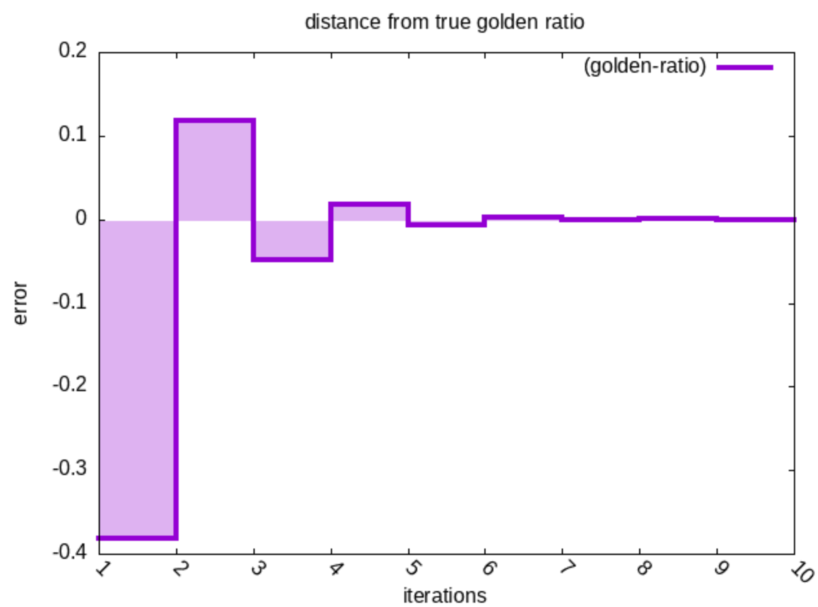
```
(cont-frac (lambda (i) 1.0)
           (lambda (i) 1.0)
           k)
```

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

ID: 169fe6a8-aba3-45c3-8e8e-181a28252426

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



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

2.55.5 Question C

ID: 8a5386a8-3be8-46b8-b9b0-e04b83df0931

> 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

ID: 1cb2b491-4ebd-4314-ac47-ac16b1126f97

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

  (rec 1))
```

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

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

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

2.56 Exercise 1.38

ID: 61db36fc-e74e-4fae-984d-648099f51a24

2.56.1 Question

ID: 7343df58-ada0-4c1c-8fb3-43433350140f

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

2.56.2 Answer

ID: eef88f33-1190-4794-b067-44c1cc80c758

```scheme
<<cont-frac>>
(define (euler k)
  (+ 2
     (cont-frac (lambda(i) 1.0)
             (lambda(i) (let ((j (1+ i)))
```

```
                        (if (= 0 (modulo j 3))
                            (* 2 (/ j 3))
                            1)))
                k)))

(euler 100)
```

```
2.7182818284590455
```

2.57 Exercise 1.39

ID: 33240ea6-67dd-4e04-a7b1-7df13beb73a2

2.57.1 Question

ID: 61af3f88-a44f-4bc8-a17f-7ea62b42b555

> 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

ID: ecd94155-42f3-434b-8c71-a16800fa6486

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

(tan-cf 55 101)
```

```
-45.1830879105221
```

2.58 1.3.4 Procedures as Returned Values

ID: 1653578d-c9b3-443c-8cc9-2495778dd9b1

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

ID: dc134e55-b99b-4cbe-accc-fd1475f484b5

2.59.1 Text

ID: 6fb6395e-367b-4503-965d-76272e2a6f71

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

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

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

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

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

2.59.2 Question

ID: 974b89d3-82b5-4c67-b67c-7f7e0341498b

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

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

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

2.59.3 Answer

ID: 28acf8cb-95d9-4517-99e1-b5497ec4b602

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

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

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

(cubic-zero 2 3 4)
```

2.60 Exercise 1.41

ID: da9d8f81-540b-4e51-922f-446e3340280b

2.60.1 Question

ID: 271f8549-974f-4b6f-8c54-314771d7ff51

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

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

2.60.2 Answer

ID: 71a9dffd-4337-4840-a923-ab857f624063

```
(define (double f)
  (lambda (x)
    (f (f x))))
```

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

21

2.61 Exercise 1.42

ID: 9eeccdb7-f70f-4832-bc89-3e6cebfc5b18

2.61.1 Question

ID: cefb2712-b83b-47bd-b455-62fc04444889

> 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

ID: ab28db3a-f050-4a37-bd07-a9862022bffb

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

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

49

2.62 Exercise 1.43

ID: 0e43bfd1-f2ae-4aa5-8821-50c89f7e72f1

2.62.1 Question

ID: 29170fd6-15da-4fe7-8a93-968bfefa43b2

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

2.62.2 Answer

ID: 71f59a74-b638-4939-b1ac-f61da9a6f79a

```
<<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

ID: 1458530a-7fc0-4686-9cc4-3296acae158a

2.63.1 Question

ID: 30dcfc41-5441-4193-9155-ab38b40430bc

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

ID: 6892b2be-25d8-4769-81ba-e563af89d2c0

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

2.64 Exercise 1.45

ID: 7c8049da-73e4-4748-903c-770016a883f7

2.64.1 Question

ID: 632c75bb-272e-4c01-85d7-1c839fcbb813

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

ID: 1f0e74e2-6ecf-4e38-8836-afa42f2471be

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

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

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

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

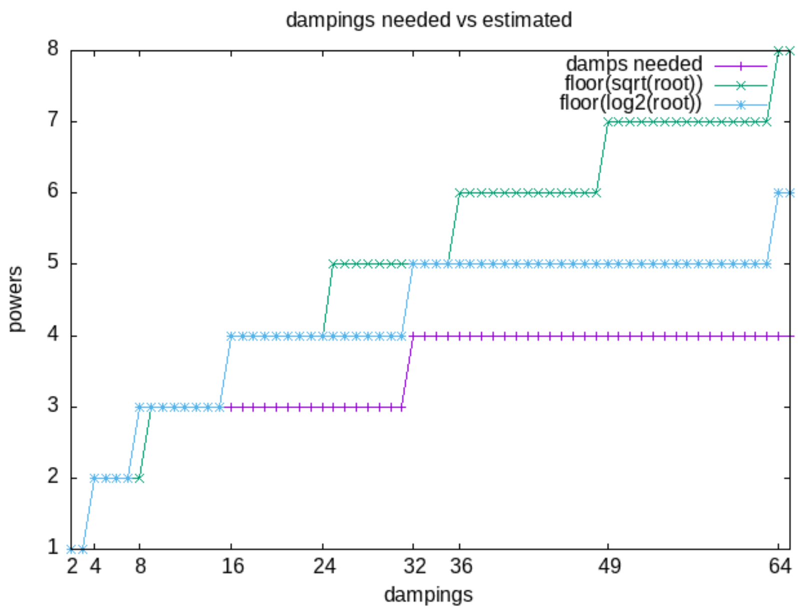
```
;; version of ”fixed-point” that will give up after a certain number of
↪  guesses.
(define (limited-fixed-point f first-guess)
  (define limit 5000)
  (define tolerance 0.00000001)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2))
```

```
        tolerance))
  (define (try guess tries)
    (if (= tries limit)
        "LIMIT REACHED"
        (let ((next (f guess)))
          (if (close-enough? guess next)
              next
              (try next (+ 1 tries))))))
    (try first-guess 1))
```

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

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

  (iter '()))
(let* ((t (reverse (nth-tester 3 65))))
  (cons '("root" "result" "damps needed" "floor(sqrt(root))"
  ↪  "floor(log2(root))")
        (map (lambda(x)
               (append x
```

```
                        (list (floor (sqrt (car x)))
                              (floor (/ (log (car x))(log 2))))))
              (map cdr t))))
```



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

2.65 Exercise 1.46

ID: e4ef33d7-c5f2-4807-9715-3c8535b6fc06

2.65.1 Question

ID: e04bcbdd-bec5-4320-81c9-5121bd1e2462

> Several of the numerical methods described in this chapter are instances of an extremely general computational strategy known as *iterative improvement*. Iterative improvement says that, to compute something, we start with an initial guess for the answer, test if the guess is good enough, and otherwise improve the guess and continue the process using the improved guess as the new guess. Write a procedure `iterative-improve` that takes two procedures as arguments: a method for telling whether a guess is good enough and a method

for improving a guess. `iterative-improve` should return as its value a procedure that takes a guess as argument and keeps improving the guess until it is good enough. Rewrite the `sqrt` procedure of 1.1.7 and the `fixed-point` procedure of 1.3.3 in terms of `iterative-improve`.

2.65.2 Answer

ID: f3669b45-83c2-4280-b4d9-c75eab7cbc96

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

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

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

```scheme
<<square>>
<<average>>
<<iterative-improve>>
(define (improve guess x)
  (average guess (/ x guess)))
(define (good-enough? next x)
  ;;   (= guess next)) ;; no longer works with this method
  (< (abs (- (square next) x)) 0.001))
(define (sqrt-improve x)
  ((iterative-improve
    (lambda(guess next)(good-enough? next x))
    (lambda(guess) (improve guess x)))
   1.0))
```

```scheme
(load "mattcheck2.scm")
<<fixed-point-txt>>
<<fixed-point-improve>>
(mattcheck "fixed-point-improve still working"
                 (fixed-point (lambda(x)(+ 1 (/ 1 x))) 1.1)
                 (fixed-point-improve (lambda(x)(+ 1 (/ 1 x))) 1.1))
<<sqrt>>
<<sqrt-improve>>
(mattcheck "sqrt-improve still working"
                 (sqrt 5)
                 (sqrt-improve 5))
```

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

3 Chapter 2: Building Abstractions with Data

ID: b89a7dcb-6526-4c26-b10f-ba6e9b63d5c9

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

ID: 442cf939-1c57-4db7-8985-5449da80b11f

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

ID: 64654350-d098-47cf-839b-faa92a13906b

3.2.1 Text

ID: c88cd22e-f8c7-47a8-9782-178c7d90c8ff

```
(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))))

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

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

```
(define (print-rat x)
  (newline)
  (display (numer x))
  (display "/")
  (display (denom x)))
```

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

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

3.2.2 Question

ID: de21a7ee-5fcf-4812-a6db-f6b71c82bdfe

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

3.2.3 Answer

ID: 2d51f2aa-f168-40b8-aea9-e5a4fc158535

```
<<abs>>
(define (make-rat n d)
  (cond ((not (or (< n 0)
                  (< d 0)))
         (cons n d))
        ((and (< n 0)
              (< d 0))
         (cons (- n) (- d)))
        (else
         (cons (- (abs n)) (abs d)))))
(define (numer x) (car x))
(define (denom x) (cdr x))

;; Bonus: an attempt to optimize
(define (make-rat-opt n d)
  (let ((nn (< n 0))
        (dn (< d 0)))
    (cond ((not (or nn dn))
           (cons n d))
          ((and nn dn)
           (cons (- n) (- d)))
          (else
           (cons (- (abs n)) (abs d))))))
```

```
<<make-rat>>
<<print-rat-txt>>
<<rat-ops-txt>>
(load "mattcheck2.scm")
(mattcheck "make-rat double negative"
           (cons 1 2)
           (make-rat -1 -2))
(mattcheck "make-rat numerator negative"
           (cons -1 2)
           (make-rat -1 2))
```

```
(mattcheck "make-rat denominator negative"
           (cons -1 2)
           (make-rat 1 -2))
(mattcheck "make-rat-opt double negative"
           (cons 1 2)
           (make-rat-opt -1 -2))
(mattcheck "make-rat-opt numerator negative"
           (cons -1 2)
           (make-rat-opt -1 2))
(mattcheck "make-rat-opt denominator negative"
           (cons -1 2)
           (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

ID: 2feac1a8-049e-4cbd-a100-8774579bc33b

3.3.1 Question

ID: 181e49cc-aeec-40f5-9ef3-7ca71b6eb372

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

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

3.3.2 Answer

ID: 11b92346-4c31-419d-8294-45ed6b89f0c7

```
<<average>>
(define (make-point x y)
  (cons x y))
(define (x-point p)
  (car p))
(define (y-point p)
  (cdr p))
(define (make-segment start end)
  (cons start end))
(define (start-segment seg)
  (car seg))
(define (end-segment seg)
  (cdr seg))
(define (midpoint-segment seg)
  (make-point (average (x-point (start-segment seg))
                       (x-point (end-segment seg)))
              (average (y-point (start-segment seg))
                       (y-point (end-segment seg)))))
(define (midpoint-segment-opt seg)
  (let ((ax (x-point (start-segment seg)))
        (bx (x-point (end-segment seg)))
        (ay (y-point (start-segment seg)))
        (by (y-point (end-segment seg))))
  (make-point (average ax
                       bx)
              (average ay
                       by))))
```

```
<<make-point>>
(load "mattcheck2.scm")
(mattcheck "make-point"
           (list 1 2)
           (let ((p (make-point 1 2)))
             (list (x-point p)
                   (y-point p))))
```

```
  (let* ((p1 (make-point 1 2))
         (p2 (make-point -1 -2))
         (s (make-segment p1 p2)))
    (mattcheck "make-segment"
               (list p1 p2)
               (list (start-segment s)
                     (end-segment s)))
    (mattcheck "midpoint-segment"
                (make-point 0 0)
                (midpoint-segment s))
    (mattcheck "midpoint-segment-opt"
                (make-point 0 0)
                (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

ID: 8d579e83-08b9-4d23-bd86-94ff1ede7ac4

3.4.1 Question

ID: ee93d34f-946b-4ea8-94c3-47762d5253b0

> 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

ID: b67b681f-b5b8-4672-befd-7cc4cea94df6

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 requested procedures first.

```scheme
(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 $D = A + (C - B) = C + (A - B)$.

```scheme
;;   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)
  (make-point (- (x-point a)
                 (x-point b))
              (- (y-point a)
                 (y-point b))))

(define (add-points a b)
  (make-point (+ (x-point a)
                 (x-point b))
              (+ (y-point a)
```

```
                 (y-point b))))

(define (rect-d R)
  (let ((a (rect-a R))
        (b (rect-b R))
        (c (rect-c R)))
    (add-points a
                (sub-points c b))))
(define (rect-d-alt R) ; should be mathematically identical.
  (let ((a (rect-a R))
        (b (rect-b R))
        (c (rect-c R)))
    (add-points c
                (sub-points a b))))

;; this is incorrect
;(define (length-points a b)
;  (let ((diffP (sub-points a b)))
;    (+ (abs (x-point diffP))
;       (abs (y-point diffP)))))
(define (length-points a b)
  (let ((ax (x-point a))
        (ay (y-point a))
        (bx (x-point b))
        (by (y-point b)))
    (sqrt (+ (square (- ax bx))
          (square (- ay by))))))

(define (rect-height R)
  (abs (length-points (rect-b R)
                 (rect-c R))))
(define (rect-width R)
  (abs (length-points (rect-b R)
                 (rect-a R))))

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

```
(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)
             (rect-area CDA)
             1)
  (mattcheck "rect-area WXYZ"
             (rect-area WXY)
             3.0)
  (mattcheck "rect-peri ABCD"
             (rect-peri ABC)
             4)
  (mattcheck "rect-peri WXYZ"
```

```
              (rect-peri WXY)
              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

ID: 3bcac536-608f-460e-8163-63298dd9963c

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.

```
<<make-point>>
;; origin is a (make-point), hwa are floats
(define (make-rect origin height width angle)
  (cons (cons origin height)
        (cons width angle)))

(define (rect-origin R)
  (caar R))
(define rect-d rect-origin)
(define (rect-height R)
  (cdar R))
(define (rect-width R)
  (cadr R))
(define (rect-angle R)
  (cddr R))

;; I underestimated how much math this would take.
(define (add-points a b)
  (make-point (+ (x-point a)
                 (x-point b))
              (+ (y-point a)
```

```
                 (y-point b))))

(define pi (* 4 (atan 1.0)))
(define (radian deg)
  (* deg (/ pi 180.0)))
(define (vector-to-xy distance angle)
      ;; rect-c: (cos(Theta),sin(Theta)) * width
      (make-point (* (cos (radian angle))
                     distance)
                  (* (sin (radian angle))
                     distance)))
      ;; could also be rotated by 90 degrees just by using
      ;;   (-sin(Theta),cos(Theta)) * height
(define (rect-c R)
  (add-points
   (rect-origin R)
   (vector-to-xy (rect-width R) (rect-angle R))))
(define (rect-a R)
  (add-points
   (rect-origin R)
   (vector-to-xy (rect-height R)
                 (+ 90 (rect-angle R)))))
(define (rect-b R)
  (add-points
   (rect-origin R)
   (add-points
    (vector-to-xy (rect-width R) (rect-angle R))
    (vector-to-xy (rect-height R)
                  (+ 90 (rect-angle R))))))
```

```
(load ”mattcheck2.scm”)

<<rect-ohwa>>
<<rect-area-peri>>

(let* ((a (make-point 13.0 14.0))
       (b (make-point 14.0 14.0))
       (c (make-point 14.0 13.0))
       (d (make-point 13.0 13.0))
       (ABC (make-rect d 1 1 0))
       (CDA (make-rect b 1 1 180))
       (w (make-point -2.0 -2.0))
       (x (make-point -2.5 1.5))
       (y (make-point -1.5 0.5))
       (z (make-point -3.0 -1.0))
       (wxy-height 1.4142135623730951)
       (wxy-width 2.1213203435596424)
       (WXY (make-rect z wxy-height wxy-width 45)))
```

```
  (mattcheck "make-rect"
             ABC
             (cons (cons d 1) (cons 1 0)))
  (mattcheck "rect-b (ABCD)"
             (rect-b ABC)
             b)
  (mattcheck "rect-b (CDAB)"
             (rect-b CDA)
             d)
  (mattcheck "rect-b (WXYZ)"
             (rect-b WXY)
             x)
  (mattcheck "rect-height"
             (rect-height WXY)
             wxy-height)
  (mattcheck "rect-width"
             (rect-width WXY)
             wxy-width)
  (mattcheck "rect-area ABCD"
             (rect-area ABC)
             (rect-area CDA)
             1)
  (mattcheck "rect-area WXYZ"
             (rect-area WXY)
             3.0)
  (mattcheck "rect-peri ABCD"
             (rect-peri ABC)
             4)
  (mattcheck "rect-peri WXYZ"
             (rect-peri WXY)
             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
SUCCEED at rect-peri WXYZ
```

3.5 2.1.3: What Is Meant by Data?

ID: 81df1f92-08c7-4b58-804a-2eb47620c500

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

ID: 367b79ac-f582-440c-8964-5df42721011e

3.6.1 Question

ID: 5205e39b-4083-4cfa-bcdf-964fac03184e

> 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

ID: 2a32f8e7-2f72-4780-ad6d-a744d6c52c27

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.

```scheme
(load "mattcheck2.scm")
<<alt-pairs-txt>>
(define (cdr z)
  (z (lambda (p q) q)))

(let ((pair (cons 0 1)))
  (mattcheck "car"
             (car pair)
             0)
  (mattcheck "cdr"
             (cdr pair)
             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

`ID: 4410574f-de61-4c0b-a1b0-d4ebdee63e9d`

3.7.1 Question

`ID: 3666d727-4d7a-4519-9a56-88c0bebd4d40`

> 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^a3^b. Give the corresponding definitions of the procedures `cons`, `car`, and `cdr`.

3.7.2 Answer

`ID: cd53185c-9159-494d-9955-078b1e8bfde6`

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`.

```
(define (cons-nnint a b)
  (* (expt 2 a) (expt 3 b)))
(define (cons-nnint-debug a b) ;; DEBUG
  (let* ((aa (expt 2 a))
         (bb (expt 3 b))
         (ab (* aa bb)))
    (display aa)
    (newline)
    (display bb)
    (newline)
    (display ab)
    (newline)
    ab))
```

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

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

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

```
<<cons-nnint>>
(let*
    ((tablesize 7)
     (inputs (map (lambda(x)
                    (map (lambda(y)
                           (cons x y))
                         (iota tablesize)))
                  (iota tablesize)))
     (outputs (map (lambda(row)
                     (map (lambda(col)
                            (cons-nnint (car col) (cdr col)))
                          row))
                   inputs)))
  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.

```
;; To find a number of some base in some column,
;; First divide by unwantedbase for targetcol number of times
<<repeated>>
(let ((targetcol 2)
      (unwantedbase 3))
  (map (lambda(row)
         (map (lambda(item)
                ((repeated (lambda(x)
                             (/ x unwantedbase)) targetcol)
                 item))
                row))
       data))
```

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

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

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

-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

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.

```
<<cons-nnint>>
<<logn>>
(use-srfis '(1))
(define (all-your-base ab unwanted wanted)
  (if (equal? (modulo ab unwanted) 0)
      (all-your-base (/ ab unwanted) unwanted wanted)
      (if (equal? (modulo ab wanted) 0)
          (round (logn wanted ab))
          ”This number isn't a factor!”)))
(define (car-nnint ab)
  (all-your-base ab 3 2))
(define (cdr-nnint ab)
  (all-your-base ab 2 3))

(let* ((initvalues '((2 3) (4 5) (7 2)))
       (conslist (map (lambda(x)
                        (apply cons-nnint x))
                      initvalues))
       (carlist (map (lambda(x)
                       (car-nnint x))
                     conslist))
       (cdrlist (map (lambda(x)
                       (cdr-nnint x))
                     conslist)))
  (map (lambda(x y) (cons x y))
       (list ”pairs” ”cons'd” ”car” ”cdr”)
       (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

ID: d2faecbf-e622-4e31-aeb7-46b6dcc2fe04

3.8.1 Question

ID: 53b69abf-b68d-488c-98e0-b925899f0119

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

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

ID: 373a5055-5305-4ba5-893e-502848a4dfae

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

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

(add-1 zero)
((lambda (f) (lambda (x)
          (f ((zero f) x)))))
((lambda (f) (lambda (x)
          (f ((lambda (x) x) x)))))
((lambda (f) (lambda (x)
          (f x))))
```

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

```scheme
(load "mattcheck2.scm")
(define one
  (lambda (f) (lambda (x)
           (f x))))
(define two
  (lambda (f) (lambda (x)
            (f (f x)))))

(mattcheck "1 = 1+0"
           1
           ((one 1+) 0))
(mattcheck "2 = 1+1+0"
           2
           ((two 1+) 0))

(define (add a b)
```

```
  (lambda (f) (lambda (x)
           ((a f) ((b f) x)))))

(mattcheck "3 = 1+2 = (1+0) + (1+1+0)"
           3
           (((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

ID: 25273dbb-b905-4c0f-82d4-019fc389114e

3.9.1 Text

ID: 860b4109-1280-4883-9562-49f57183f067

```
(define (add-interval x y)
  (make-interval (+ (lower-bound x) (lower-bound y))
                 (+ (upper-bound x) (upper-bound y))))
(define (mul-interval x y)
  (let ((p1 (* (lower-bound x) (lower-bound y)))
        (p2 (* (lower-bound x) (upper-bound y)))
        (p3 (* (upper-bound x) (lower-bound y)))
        (p4 (* (upper-bound x) (upper-bound y))))
    (make-interval (min p1 p2 p3 p4)
                   (max p1 p2 p3 p4))))
(define (div-interval x y)
  (mul-interval
   x
   (make-interval (/ 1.0 (upper-bound y))
                  (/ 1.0 (lower-bound y)))))
```

3.9.2 Question

ID: 77308977-b69d-4fd5-8c1b-cadd954f689c

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

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

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

3.9.3 Answer

ID: 6468e6ee-f282-406a-9b12-7c2f767013eb

```
<<interval-txt>>

;; Makes more sense to me to test
;; order in the constructor than selector
(define (make-interval a b)
  (if (> a b)
      (cons a b)
      (cons b a)))

(define (upper-bound i)
  (car i))
(define (lower-bound i)
  (cdr i))
```

3.10 Exercise 2.8

ID: fc6580f7-1084-45af-be42-d838bc1e7686

3.10.1 Question

ID: 691b6556-635f-4283-bd5e-45a6bc2f005a

> 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

ID: 58aa4210-0605-4053-a0bd-14246bc47115

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

ID: f10d8027-e143-4262-b9e0-0e8607f4e240

3.11.1 Question

ID: 349cd4e2-bf0d-496a-b7a3-761deeed95e7

> 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

ID: 922c3352-632d-4531-915a-7385bd10f1dd

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:

```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))
       (Mab (mul-interval ia ib))
       (Dab (div-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 addition is distributive"
                     (width-interval Aab)
                     (+ (width-interval ia)
                        (width-interval ib)))
    (mattcheck-float "width subtraction is distributive"
                     (width-interval Sab)
                     (- (width-interval ia)
                        (width-interval ib)))
    (mattcheck-float "width multiplication is distributive"
                     (width-interval Mab)
                     (* (width-interval ia)
                        (width-interval ib)))
    (mattcheck-float "width division is distributive"
                     (width-interval Dab)
                     (/ (width-interval ia)
                        (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: 0.019999999999999995
returned: 2.5
FAIL at width division is distributive
expected: 0.49999999999999978
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 $z_{width} = x_{width} + y_{width}$
> Multiplied or divided widths cannot be determined from widths alone.

So, let's try that again.

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

```scheme

(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

ID: b9af79c4-abc2-473b-9973-bf0ecc5e036f

3.12.1 Question

ID: c1ab865c-3dfc-4656-a2f6-ffd4e42df260

> 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.

3.12.2 Answer

ID: ec0d39fc-09ca-4311-8742-6ba091b2568d

```
(define (interval-spans-zero? I)
  (and (> upper-bound 0)
       (< lower-bound 0)))
(define (div-interval x y)
  (if (interval-spans-zero? y)
      "DIV-INTERVAL ERROR: denominator spans zero"
      (mul-interval
       x
       (make-interval (/ 1.0 (upper-bound y))
                      (/ 1.0 (lower-bound y))))))
```

3.13 Exercise 2.11

ID: fe8157f6-0e9b-4048-8f97-837cfb1797bb

3.13.1 Question

ID: be471a80-4642-4792-ba1e-cdc4cd0ae5e3

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

3.13.2 Answer

ID: 56419705-45bd-4fdc-8f4c-478f840a1751

This problem doesn't appear to have a beautiful, elegant answer. Let's examine the nine cases.

```
(use-modules (ice-9 format))
(use-srfis '(1))
(load "mattcheck2.scm")
<<make-interval>>
<<sub-interval>>

(define (matt-examine-mult f)
  (let* ((pp (make-interval 3 2))
         (pn (make-interval 3 -5))
         (pn2 (make-interval 1 -0.5))
         (nn (make-interval -5 -7))
         (listofpairs (list
                       (list pp pp)
                       (list pp pn)
                       (list pp nn)
                       (list pn pp)
                       (list pn pn)
```

```
                       (list pn pn2) ;;<- edge case to catch incomplete
                       (list pn nn)  ;;   multiplication functions
                       (list nn pp)
                       (list nn pn)
                       (list nn nn)))
         (givesign (lambda(x)
                     (if (negative? x)
                         "-"
                         "+")))
         (print-sign (lambda(I)
                       (format #f "~a ~a"
                               (givesign (upper-bound I))
                               (givesign (lower-bound I)))))
         (print-int (lambda(I)
                      (format #f "~a/~a"
                              (upper-bound I)
                              (lower-bound I))))
         (print-ints (lambda(I J)
                       (format #f "~a times ~a"
                               (print-int I)
                               (print-int J))))
         (results (map (lambda(p)
                         (apply f p))
                       listofpairs)))
    (list
     (map (lambda(p)
            (apply print-ints p))
          listofpairs)
     (map print-int results)
     (map (lambda(I)
            (print-sign I))
          results)
     (map (lambda(p)
            (format #f "~a // ~a"
                    (print-sign (car p))
                    (print-sign (cadr p))))
          listofpairs))))
(cons
 (list "problem" "result" "signs" "problem signs")
 (apply zip
        (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	+ +	- - // - -

```
(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)))
```

```scheme
               (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")
<<make-interval>>
<<sub-interval>>
<<mul-interval-opt>>

(define (matt-mult-consistency f1 f2)
  (let* ((pp (make-interval 3 2))
         (pn (make-interval 3 -5))
         (pn2 (make-interval 1 -0.5))
         (nn (make-interval -5 -7))
         (listofpairs (list
                       (list "pp*nn" pp pp)
                       (list "pp*pn" pp pn)
                       (list "pp*nn" pp nn)
                       (list "pn*pp" pn pp)
                       (list "pn*pn" pn pn)
                       (list "pn*pn2" pn pn2) ;;<- edge case to catch incomplete
                       (list "pn*nn" pn nn)  ;;   multiplication functions
                       (list "nn*pp" nn pp)
                       (list "nn*pn" nn pn)
                       (list "nn*nn" nn nn))))
    (map (lambda(l)
           (mattcheck (car l)
                      (apply f1 (cdr l))
                      (apply f2 (cdr l))))
         listofpairs)))

(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

ID: ccd05e64-baed-4c60-8cef-c767e30e471e

3.14.1 Question

ID: df7c487f-db1f-4e9a-ac3c-3f4669c0fb96

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

ID: 5ae1028e-a8e1-4b7c-841b-6e0a963b8587

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

```scheme
<<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))))
```

```scheme
<<interval-percent>>
(load "mattcheck2.scm")
  (define (roughly-eq? a b)
    ;; error size varies with magnitude of fp
    ;; so dx must vary too.
    (define dx (* a 0.000001))
    (and (> a (- b dx))
         (< a (+ b dx))))
(define (interval-roughly-eq? I J)
  (and (roughly-eq? (upper-bound I) (upper-bound J))
       (roughly-eq? (lower-bound I) (lower-bound J))))

(let* ((i1 (make-interval 105.0 95.0))
       (i2 (make-center-width 100.0 5))
       (i3 (make-center-percent 100.0 5))
       (i1a (upper-bound i1)))
  (mattcheck "make-center-width"
             i1
             i2)
  (mattcheck "make-center-percent"
             i1
             i3)
  (mattcheck "percent"
             (percent i1)
             (percent i3)
             5.0)
  (mattcheck+ "make-center-percent is consistent"
              (list i1 i3)
              #: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

ID: 3280cdb0-2f9f-4ede-9f29-0c46af1e326e

3.15.1 Question

ID: 7e2dfdf8-2ad3-4882-b14e-0f2c450bffa2

> 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

ID: 54bf6179-da7c-4543-a652-186d01cf3e62

I should've written this function a while ago.

```
(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.

```
<<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)?

```
<<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

ID: d1e66abc-2f0f-4cc6-907d-309e34fda663

3.16.1 Question

ID: c49a3157-62af-4e7a-81c1-51b9df4e292e

> 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:

```
(define (par1 r1 r2)
  (div-interval (mul-interval r1 r2)
                (add-interval r1 r2)))

(define (par2 r1 r2)
  (let ((one (make-interval 1 1)))
    (div-interval
     one (add-interval (div-interval one r1)
                       (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

ID: a1f25a4b-7a82-4715-b5bb-c839dc628ac4

```
<<echo>>
<<interval-percent>>
<<mul-interval-opt>>
<<par-resistors>>

(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

ID: c7eae4d2-1154-4308-b73d-02a1da53195a

3.17.1 Question

ID: 2c1f64e2-c811-4ae2-ad5f-94c59005218c

> 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

ID: 189f4ab1-0028-46f2-8a06-a33a9b40e08f

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:

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

(define A (make-center-percent 10 1))
(echo "A+A = 2A !=" (add-interval A A))
(echo "A-A = 0 !=" (sub-interval A A))
(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

ID: b8b8d432-47ab-4218-8bad-d6fa6a70fde5

3.18.1 Question

ID: 9ef9c5f1-107f-4e7a-a59a-34074832cdf9

> 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

`ID: fa0bb72f-72e1-44bb-a86c-b27774b9f9da`

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

`ID: 83758def-8586-4f95-a292-e262eeec307c`

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

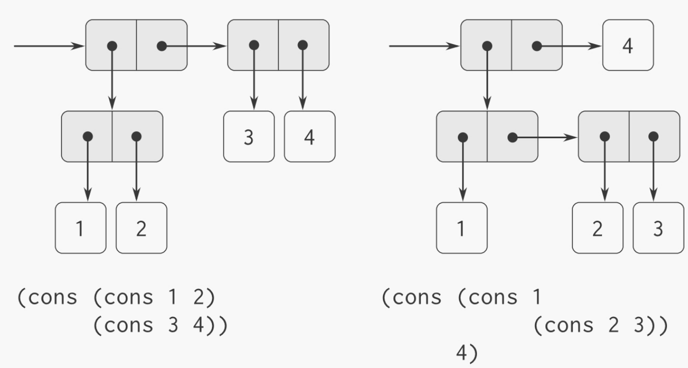


Figure 2.3: Two ways to combine 1, 2, 3, and 4 using pairs.

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

ID: a17be458-3194-46f1-9961-2aa4c052f60f

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

ID: df832283-cdbb-40a3-8536-5e029fd3f4dd

3.21.1 Question

ID: e8fd5c2d-cee3-4178-badf-230243451076

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

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

3.21.2 Answer

ID: 0dec0924-7246-4528-98a2-5b137ec500a2

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

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

| 34 |

3.22 Exercise 2.18

ID: 613581ea-f114-42c7-8179-cc85ad288d59

3.22.1 Question

ID: aac18e73-5930-4ed9-b48a-82aab894bd19

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

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

3.22.2 Answer

ID: ac51a6b8-4121-406d-bc54-f3a394594a56

```scheme
(define (reverse l)
  (define len (length l))
  (define (iter i result)
    (if (< (1- len) i)
        result
        (iter (+ i 1)
              (cons (list-ref l i)
```

```
                    result))))
  (iter 0 '()))
```

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

34 149 72 23

3.23 Exercise 2.19

ID: 49dafc3f-cfe8-4bf5-9612-d75c6b7c75c1

3.23.1 Question

ID: 66168120-ab19-4de4-a7ed-3f66bf6aee39

> 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:

```scheme
(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

ID: 349c968a-5e4d-450f-95da-1e5bccd3ce18

```scheme
<<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)
(define except-first-denomination cdr)
(define no-more? null?)
```

```scheme
<<Ex-2-19>>
(list
 (cc 100 us-coins)
 (cc 100 (reverse us-coins))
 (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

ID: 3fddccfe-0012-4058-b840-0617fdbe104f

3.24.1 Question

ID: 28845497-e50d-434b-a37e-f5f8274b8998

> 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,

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

3.24.2 Answer

ID: 884ab392-ed28-4843-aa00-0eea45c63543

```
(define (same-parity . rest)
  (define same?
    (if (even? (car rest))
        even?
        odd?))
  (define (iter l results)
    (if (null? l)
         results
        (let ((a (car l)))
          (iter (cdr l)
                (if (same? a)
                    (cons a results)
                    results)))))
  (iter (reverse rest) '()))

;; Attempting to remove the reversing
(define (same-parity2 . args)
  (define first (car args))
  (define same?
    (if (even? first)
        even?
        odd?))
  (define (iter l results)
    (if (null? l)
        results
        (let ((a (car l))
              (d (cdr l)))
```

```
          (if (same? a)
              (iter d (append results
                              (cons a #nil)))
              (iter d results)))))
  (iter (cdr args) (cons first #nil)))
```

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

ID: 6a0ed282-f958-4baa-a0a4-9cf112a72a0d

3.25.1 Question

ID: 39acd29c-4385-4244-a4b8-ca659597d5ea

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

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

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

```
(define (square-list items)
  (if (null? items)
      nil
      (cons <??> <??>)))
(define (square-list items)
  (map <??> <??>))
```

3.25.2 Answer

ID: 39d96109-1859-49c7-8fcd-9dff4d8fc4b0

```
<<square>>
(define (square-list-manual items)
  (if (null? items)
      #nil
      (cons (square (car items))
            (square-list-manual (cdr items)))))
(define (square-list items)
  (map square items))
```

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

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

3.26 Exercise 2.22

ID: 4757f6ab-7e16-4169-8a77-9f42c2747abe

3.26.1 Questions

ID: 704bb669-f8e9-47db-bab6-0c49bb8c3c77

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

```
(define (square-list items)
  (define (iter things answer)
   (if (null? things)
       answer
       (iter (cdr things)
             (cons (square (car things))
                   answer))))
  (iter items nil))
```

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

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

```scheme
(define (square-list items)
  (define (iter things answer)
    (if (null? things)
        answer
        (iter (cdr things)
              (cons answer
                    (square (car things))))))
  (iter items nil))
```

This doesn't work either. Explain.

3.26.2 Answer

ID: d4f705d9-a46f-4726-8a36-6b161fcf3845

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:

```scheme
(square-list (iota 6))
(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:

```scheme
(square-list (iota 6))
((((((#nil . 0) . 1) . 4) . 9) . 16) . 25)
;; Equivalent to:
(cons (cons (cons (cons (cons (cons #nil
                                    0)
                              1)
                        4)
                  9)
            16)
      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.

3.27 Exercise 2.23

ID: 9931dcde-c75d-4277-916d-e26d012aa3cc

3.27.1 Question

ID: 3252b8cc-e1ab-4855-808f-a383e27a7396

> 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-each` is used with procedures that perform an action, such as printing. For example,

```
(for-each (lambda (x)
            (newline)
            (display x))
          (list 57 321 88))
;; 57
;; 321
;; 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

ID: d98e6318-8fd5-4c0f-8943-734a0eb6d108

```
(define (for-each-mine proc items)
  (define (iter l)
    (if (null? l)
        #t
        (begin (proc (car l))
               (iter (cdr l)))))
  (iter items))
```

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

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

3.28 Exercise 2.24

ID: 1859958b-e5fe-45e1-9c88-1bd50d0bccfc

3.28.1 Text Definitions

ID: 75e2bc99-9250-45c5-bc99-68e9f7435478

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

3.28.2 Question

ID: 2fc07509-94f8-4057-af0a-57e1975cca88

> 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

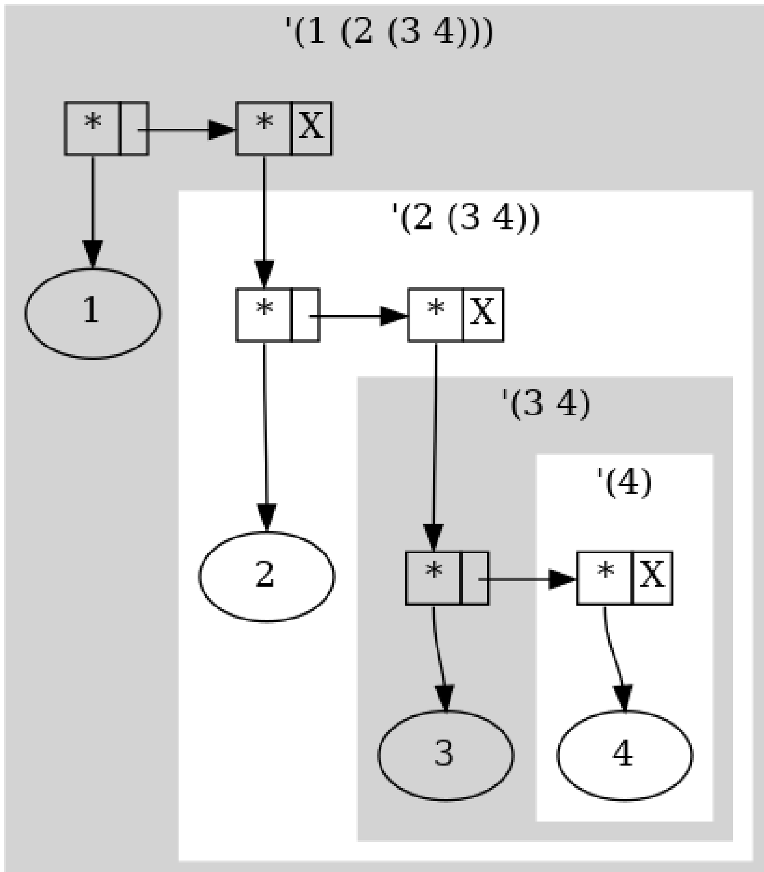
ID: 5053c146-9dbe-470c-bc7b-505bf646f302

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:

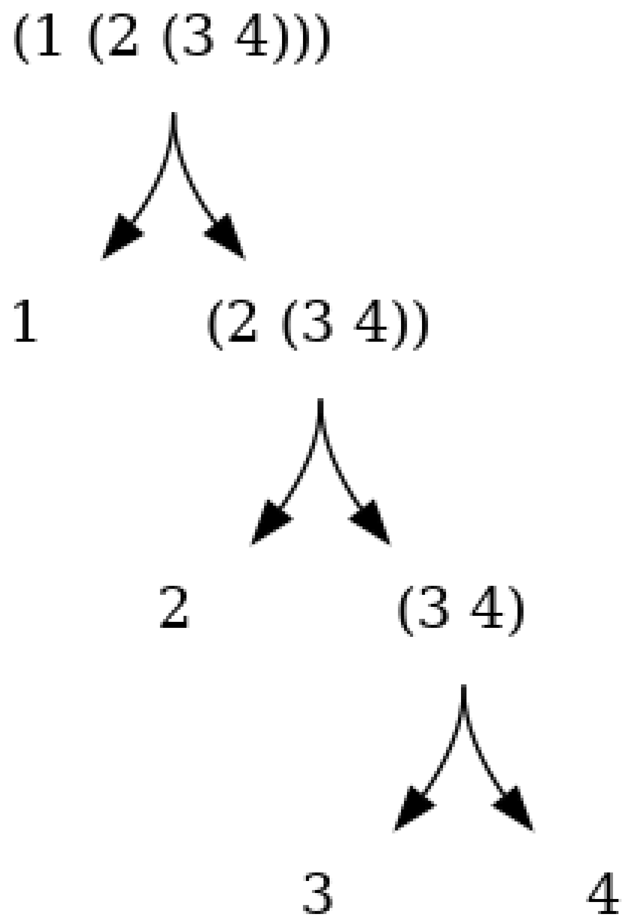
```
<<echo>>
(let ((l1 (list 1 (list 2 (list 3 4))))
      (l2 (cons 1
                (cons
                 (cons 2
                       (cons
                        (cons 3
                              (cons 4
                                    #nil))
                        #nil))
                 #nil))))
  (echo "textbook version:" l1)
  (echo "cons'd version:" l2))
```

```
textbook version: (1 (2 (3 4)))
cons'd version: (1 (2 (3 4)))
```

Dot and box version:



Tree version:



3.29 Exercise 2.25

ID: ee1f06c2-8b9e-45e1-b411-f3d55ed81d4b

3.29.1 Question

ID: 73922dc1-c1cf-4667-a652-be1c4562b1fa

> 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))))))
```

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

```
7 7 7
```

3.30 Exercise 2.26

ID: 2420a238-983e-45b6-96f8-200c89225ea8

3.30.1 Question

ID: 3e049c8e-7c19-4260-9719-23aff843ee1c

Suppose we define `x` and `y` to be two lists:

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

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

```
(append x y)
(cons x y)
(list x y)
```

3.30.2 Answer

ID: 43f5805e-5b76-43e0-a645-d9d591c4c480

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

```
  (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))
```

3.31 Exercise 2.27

ID: 54e96be8-2cea-4976-867d-43dd31887156

3.31.1 Question

ID: 3eff3209-704b-42a7-8033-f43e4520f409

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

ID: 47d43f92-a841-4bb6-84d7-6b13036fddf6

```
(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

ID: e4661795-7d35-4b4c-87c8-4f27e61504e6

3.32.1 Question

ID: 1aa047af-6107-444f-9460-4720db3d56b5

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

3.32.2 Answer

ID: 03543166-fce5-48d5-b81b-447c9a945bcb

```
(define (fringe l)
  (if (null? l)
      #nil
      (let ((a (car l))
            (d (cdr l)))
        (append (if (pair? a)
                    (fringe a)
                    (list a))
                (fringe d)))))
```

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

```
(1 2 3 4 5)
```

3.33 Exercise 2.29: Binary Mobiles

ID: 63a9edde-9764-4d9e-8100-baed248c2dcb

3.33.1 Text Definitions

ID: f0163b74-c449-4a13-ad88-1da47321397a

```
(define (make-mobile left right)
  (list left right))

(define (make-branch length structure)
  (list length structure))
```

3.33.2 Question A: Selectors

ID: 20aad58f-3c4f-4fa8-8d27-4a4a5aee09b3

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

3.33.3 Answer A

ID: 9412f002-9e0b-4e44-8bb4-195154f0b0d7

```
<<mobile-constructors-list>>
(define (left-branch mobile)
  (car mobile))
(define (right-branch mobile)
  (cadr mobile))
(define (branch-length branch)
  (car branch))
(define (branch-structure branch)
  (cadr branch))
```

3.33.4 Question B: total-weight

ID: 41775c86-5d20-49c2-9030-d07fc6174431

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

3.33.5 Answer B

ID: 6b9e8068-4729-464e-b963-f0ddfa5c0f64

```
(define (total-weight mobile)
  (let ((leftS (branch-structure (left-branch mobile)))
        (rightS (branch-structure (right-branch mobile))))
    (+ (if (number? leftS)
           leftS
```

```
          (total-weight leftS))
      (if (number? rightS)
          rightS
          (total-weight rightS)))))
```

```
<<mobile-selectors-list>>
<<mobile-total-weight>>

(let ((M1 (make-mobile
           (make-branch 5 5)
           (make-branch 1
                        (make-mobile (make-branch 2 2)
                                     (make-branch 2 3)))))
      (M2 (make-mobile
           (make-branch 2
                        (make-mobile
                         (make-branch 2 2)
                         (make-branch 2 2)))
           (make-branch 2
                        (make-mobile
                         (make-branch 2 2)
                         (make-branch 2 2))))))
  (list (total-weight M1)
        (total-weight M2)))
```

10	8

3.33.6 Question C: Balancing

ID: 2137f250-1d72-4cf9-becc-26f1a07c4a61

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

ID: f2cad0b2-6145-4224-8476-9e217f12028f

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.

```
(define (total-torque branch)
  (let ((len (branch-length branch))
        (struct (branch-structure branch)))
    (* len
```

```
      (if (number? struct)
          struct
          (let ((lbs (branch-structure (left-branch struct)))
                (rbs (branch-structure (right-branch struct))))
            (+ (total-weight lbs)
               (total-weight rbs)))))))
(define (balanced? mobile)
  (let* ((l (left-branch mobile))
         (ls (branch-structure l))
         (r (right-branch mobile))
         (rs (branch-structure r))
         (l-balanced (if (number? ls)
                         #t
                         (balanced? ls)))
         (r-balanced (if (number? rs)
                         #t ;; Fixed: accidentally used ls again.
                         (balanced? rs))))
    (if (and l-balanced r-balanced)
        (= (total-torque l)
           (total-torque r))
        #f)))
```

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

```
(define (total-weight mobile)
  (if (number? mobile)
      mobile ;; this is a weight, just return it
      (let ((leftS (branch-structure (left-branch mobile)))
            (rightS (branch-structure (right-branch mobile))))
        (+ (if (number? leftS)
               leftS
               (total-weight leftS))
           (if (number? rightS)
               rightS
               (total-weight rightS))))))
```

```
<<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))
 (isbalanced? "M3" (balanced? M3))
 (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

ID: f5a9dc34-2ad7-4c13-944c-d0231cab3e7f

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

```
(define (make-mobile left right)
  (cons left right))

(define (make-branch length structure)
  (cons length structure))
```

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

3.33.9 Answer D

ID: ff3509a0-b674-4752-9abd-b823d35fed43

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

```
<<mobile-constructors-cons>>
(define (left-branch mobile)
  (car mobile))
(define (right-branch mobile)
  (cdr mobile))
(define (branch-length branch)
  (car branch))
(define (branch-structure branch)
  (cdr branch))
```

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

```
<<echo>>
<<mobile-selectors-cons>>
<<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))
(isbalanced? ”M3” (balanced? M3))
(isbalanced? ”M4” (balanced? M4))
```

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

3.34 Exercise 2.30

ID: 8fb7c1b7-31c5-4f08-8cf6-12094e850c3c

3.34.1 Question

ID: 36f8065d-dd3b-4120-851e-89c04ae7ffd4

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

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

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

3.34.2 Answer

ID: 203d3de0-6067-4723-a81d-bf43e4a3cb64

```
<<square>>
(define (square-tree-discrete tree)
  (cond ((null? tree) '())
        ((not (pair? tree)) (square tree))
        (else (cons (square-tree-discrete (car tree))
                    (square-tree-discrete (cdr tree))))))

(define (square-tree-map tree)
  (map (lambda (sub-tree)
         (if (pair? sub-tree)
             (square-tree-map sub-tree)
             (square sub-tree)))
       tree))
```

```
(load "mattcheck2.scm")
<<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-discrete"
             (square-tree-discrete testlist)
             answer)
  (mattcheck "square-tree-map"
```

```
            (square-tree-map testlist)
            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`.

```
(let ((parens-list (cons 1 (cons 2 (cons 3 '()))))
      (nil-list (cons 1 (cons 2 (cons 3 #nil)))))
  (display parens-list)(display " <== ends with '()")
  (newline)
  (display nil-list)(display " <== ends with #nil")
  (newline)
  (display "Are these two lists equal? > ")
  (display (equal? parens-list nil-list))
  (newline)
  (display "Does Guile consider #nil and '() equal? > ")
  (display (equal? #nil '()))
  (newline)
  (display "What about #nil and #f? > ")
  (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

ID: 4ddf43c7-fd1b-441d-a599-0a8412646c1f

3.35.1 Question

ID: 5d38b71c-3887-48f7-b978-e90a17b6fbde

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

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

3.35.2 Answer

ID: e70b89fa-ced4-4294-bf97-93177b5296bf

```scheme
(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))))))
```

```scheme
(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

ID: fb825e93-a665-49dd-b425-5c08f0a176b4

3.36.1 Question

ID: d367324d-0a43-4131-87e5-4789a887d8ae

> 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:

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

3.36.2 Answer

ID: 1c89de82-54ce-48f2-8ac3-2382391211fc

```
(define (subsets s)
  (if (null? s)
      (list '())
      (let ((rest (subsets (cdr s))))
        (append rest (map (lambda(x)
                            (cons (car s) x))
                          rest)))))
```

```
<<subsets>>
(load "mattcheck2.scm")
(let ((answer
       (list '()
             (list 3)
             (list 2) (list 2 3)
             (list 1) (list 1 3)
             (list 1 2) (list 1 2 3))))
  (mattcheck "subsets"
             (subsets (list 1 2 3))
             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.

```
<<echo>>
(define (doit f x)
  (f x) ;; this probably has a formal lambda calculus name
  x)    ;; but I don't know what it is
(define (echo-return x)
  (doit echo x))
(define (echo-y-return-x x . y)
  (apply echo y)
  x)
```

```
<<subsets>>
<<echo-return>>
(define (subsets-debug s)
  (echo "Enter with" s "{")
  (let ((a1
         (if (null? s)
             (echo-return (list '()))
             (let ((rest (subsets-debug (cdr s))))
               (echo "[ iter" (car s) "over" rest "]")
               (append rest (map (lambda(x)
                                   (let ((y (cons (car s) x)))
                                     (format #t "~a " y)
                                     y))
                                 rest))))))
    (echo "} end" s)
    a1))
(let ((answer
       (list '()
             (list 3)
             (list 2) (list 2 3)
             (list 1) (list 1 3)
             (list 1 2) (list 1 2 3))))
  (subsets-debug (list 1 2 3))
  answer)
```

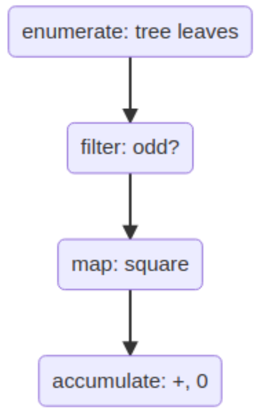
```
Enter with (1 2 3) {
Enter with (2 3) {
Enter with (3) {
Enter with () {
(())
} end ()
[ iter 3 over (()) ]
(3)
} end (3)
[ iter 2 over (() (3)) ]
(2) (2 3)
} end (2 3)
[ iter 1 over (() (3) (2) (2 3)) ]
(1) (1 3) (1 2) (1 2 3)
} end (1 2 3)
```

3.37 2.2.3: Sequences as Conventional Interfaces

ID: ae4c378f-a18e-4988-9dfd-7c5404e78c9e

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:



3.38 Exercise 2.33: The flexibility of `accumulate`

ID: 36c3ba68-f680-4525-9dab-23de05e01770

3.38.1 Text Definitions

ID: 3095b036-e7c3-42d7-a386-d352cb3abd2e

```
(define (accumulate op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence)
          (accumulate op
                      initial
                      (cdr sequence)))))
```

3.38.2 Question

ID: 2c3c90cc-d492-470b-b625-372f136c6dff

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

```
(define (map p sequence)
  (accumulate (lambda (x y) <??>) nil sequence))
(define (append seq1 seq2)
  (accumulate cons <??> <??>))
(define (length sequence)
```

```
  (accumulate <??> 0 sequence))
```

3.38.3 Answer

ID: 7785a981-acf8-4b1a-8f5b-77ca062c43a0

```
(define (map-acc p sequence)
  (accumulate (lambda (x y)
                (cons (p x) y))
              '() sequence))
(define (append-acc seq1 seq2)
  (accumulate cons seq2 seq1))
(define (length-acc sequence)
  (accumulate (lambda(x y)
                (1+ y))
              0 sequence))
```

```
(load "mattcheck2.scm")
<<accumulate>>
<<accumulate-forms>>
<<square>>
(let ((l (list 1 2 3 4)))
  (mattcheck "map"
             (map square l)
             (map-acc square l))
  (mattcheck "append"
             (append l l)
             (append-acc l l))
  (mattcheck "length"
             (length l)
             (length-acc l)))
```

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

3.39 Exercise 2.34

ID: 70003a6c-ebeb-4a1a-a29a-c637af29e348

3.39.1 Question

ID: 20b40a17-9a77-43fe-bc32-3cd9a590f447

> 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.

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

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

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

3.39.2 Answer

ID: 4f30d8db-3295-4190-bf1f-8fbcba81c950

```
(define (horner-eval x coefficient-sequence)
  (accumulate (lambda (this-coeff higher-terms)
                (+ this-coeff
                   (* higher-terms x)))
              0
              coefficient-sequence))
```

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

ID: d2ac02f3-e7d4-41d4-9265-5272695dbd92

3.40.1 Question

ID: fad0445e-b03b-4996-bf19-be6d81e84afd

Redefine `count-leaves` from 2.2.2 as an accumulation:

3.40.2 Answer

ID: 12b8a86f-f827-4522-a662-b38513d09b7b

```
(define (count-leaves-acc t)
  (accumulate (lambda(i total)
                (+ i total))
              0 (map (lambda(x)
                       (if (pair? x)
                           (count-leaves-acc x)
                           1))
                     t)))
```

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

SUCCEED at count-leaves-acc

3.41 Exercise 2.36: Accumulate across multiple lists

ID: 075ed620-b106-4eca-b1cf-185c181e032e

3.41.1 Question

ID: fb101d9b-4436-4566-9c20-32ddd0a4bb0c

> 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`:

```scheme
(define (accumulate-n op init seqs)
  (if (null? (car seqs))
      nil
      (cons (accumulate op init <??>)
            (accumulate-n op init <??>))))
```

3.41.2 Answers

ID: 29bb2475-8f2b-46b0-8cc3-c5f5ea0d98dc

```scheme
(define (accumulate-n op init seqs)
  (if (null? (car seqs))
      '()
      (cons (accumulate op init
                        (map car seqs))
            (accumulate-n op init
                          (map cdr seqs)))))
```

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

```
SUCCEED at accumulate-n
```

3.42 Exercise 2.37: Enter the matrices

ID: 14ee47b2-bdb6-44f1-806e-0ce544f38cae

3.42.1 Question

ID: aa6cbc48-348d-485c-8973-8d3151d7ad26

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:

```
(define (dot-product v w)
  (accumulate + 0 (map * v w)))
(define (matrix-*-vector m v)
  (map <??> m))
(define (transpose mat)
  (accumulate-n <??> <??> mat))
(define (matrix-*-matrix m n)
  (let ((cols (transpose n)))
    (map <??> m)))
```

3.42.2 Answer

ID: d38d82ff-dd0f-4555-841a-1a66d8fbacf3

```
(define (dot-product v w)
  (accumulate + 0 (map * v w)))
(define (matrix-*-vector m v)
  (map (lambda(row)
         (dot-product v row))
       m))
(define (transpose mat)
  (accumulate-n cons '() mat))
(define (matrix-*-matrix m n)
  (let ((cols (transpose n)))
    (map (lambda(row)
           (map (lambda(col)
                  (dot-product row col))
                cols))
           m)))
```

```
(load "mattcheck2.scm")
<<echo>>
<<accumulate>>
<<accumulate-n>>
```

```
<<matrix-ops>>
(let* ((v1 (list 1 2 3 4))
       (v2 (list 5 6 7 8))
       (m1 (list v1 v2))
       (m1t (list (list 1 5)
                  (list 2 6)
                  (list 3 7)
                  (list 4 8)))
       (m1-*-m1t (list (list 30 70)
                       (list 70 174))))
  (mattcheck "transpose"
             (transpose m1)
             m1t)
  (mattcheck "dot-product"
             (dot-product v1 v2)
             70)
  (mattcheck "matrix-*-vector"
             (matrix-*-vector m1 v1)
             (list 30 70))
  (mattcheck "matrix-*-matrix"
             (matrix-*-matrix m1 m1t)
             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:

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

But much harder to comprehend Lisp code like this:

```
(map (lambda(row)
       (map (lambda(col)
              (dot-product row col))
            m1t))
```

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

```
    m1)
```

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

3.43 Exercise 2.38: fold-right

ID: e718c293-a8c8-4c16-8ee9-7d108bab3fa0

3.43.1 Question A

ID: 68e456e3-b545-438a-850f-634035a184e8

> 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:

```
(define (fold-left op initial sequence)
  (define (iter result rest)
    (if (null? rest)
        result
        (iter (op result (car rest))
              (cdr rest))))
  (iter initial sequence))
```

> What are the values of

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

3.43.2 Answer A

ID: 8a5a2b11-b9f7-448d-8690-32b88fea7085

```
<<accumulate>>
(define fold-right accumulate)
<<fold-left>>
<<echo>>
(echo "(fold-right / 1 (list 1 2 3)):" (fold-right / 1 (list 1 2 3)))
(echo "(fold-left / 1 (list 1 2 3)):" (fold-left / 1 (list 1 2 3)))
(echo "(fold-right list nil (list 1 2 3))" (fold-right list '() (list 1
↪  2 3)))
```

```
(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

ID: aebf1d56-1a8f-4519-8d3d-7bf1854718f2

> 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

ID: cd5db947-e91d-4200-a5ba-69b708c6ae62

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

```
(load "mattcheck2.scm")
<<accumulate>>
(define fold-right accumulate)
<<fold-left>>
<<echo>>

(let* ((l (list 1 2 3 4 5))
       (lr (reverse l)))
  (mattcheck "commutative addition"
             (fold-right + 0 l)
             (fold-right + 0 lr)
             (fold-left + 0 l)
             (fold-left + 0 lr))
  (mattcheck "commutative multiplication"
             (fold-right * 1 l)
             (fold-right * 1 lr)
             (fold-left * 1 l)
             (fold-left * 1 lr)))
```

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

3.44 Exercise 2.39: reverse via fold

ID: 65c7d1f3-bd94-4603-9534-3028979dd321

3.44.1 Question

ID: ad8bef77-fb45-4d6b-997a-2263472c20c3

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

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

3.44.2 Answer

ID: 3b6f3fc0-a210-4091-b521-93ce82a913ba

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

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

Now to the problem.

```
<<fold-compat>>
(define (reverse-fr sequence)
  (fold-right (lambda (x y)
                (append  y (list x)))
              '() sequence))
(define (reverse-fl sequence)
  (fold-left (lambda (x y)
               (append (list x) y))
             '() sequence))
```

```
(load "mattcheck2.scm")
<<reverse-fold>>
(let ((l (iota 5))
      (lr (reverse (iota 5))))
  (mattcheck "reverse-fr"
             (reverse-fr l)
             lr)
  (mattcheck "reverse-fl"
             (reverse-fl l)
             lr))
```

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

3.45 Exercise 2.40: `unique-pairs`

ID: 8fbcce71-99a3-4265-9e3c-4edee2d0c972

3.45.1 Text Definitions

ID: 04c72f6b-beb5-4008-8e30-e706e318a5b3

```scheme
(define (enumerate-interval low high)
  (if (> low high)
      '()
      (cons low
            (enumerate-interval
             (+ low 1)
             high))))
```

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

```scheme
<<flatmap>>
<<enumerate-interval>>
<<prime-smallest-divisor>>
(define (prime-sum? pair)
  (prime? (+ (car pair) (cadr pair))))
(define (make-pair-sum pair)
  (list (car pair) (cadr pair) (+ (car pair) (cadr pair))))
(define (prime-sum-pairs n)
  (map make-pair-sum
       (filter prime-sum? (flatmap
                           (lambda (i)
                             (map (lambda (j) (list i j))
                                  (enumerate-interval 1 (- i 1))))
                           (enumerate-interval 1 n)))))
```

3.45.2 Question

ID: 5b21dfdd-eb13-4cec-ae85-79eb0d3fbcf3

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

3.45.3 Answer

ID: f5dfcaf8-f191-4701-9807-6962df851c3e

```scheme
<<fold-compat>>
<<enumerate-interval>>
<<flatmap>>
(define (unique-pairs n)
  (flatmap
   (lambda (i)
     (map (lambda (j) (list i j))
          (enumerate-interval 1 (- i 1))))
   (enumerate-interval 1 n)))
```

```scheme
(load "mattcheck2.scm")
<<unique-pairs>>
(let ((target 5)
      (answer (list (list 2 1)(list 3 1)(list 3 2)(list 4 1)
                    (list 4 2)(list 4 3)(list 5 1)(list 5 2)
                    (list 5 3)(list 5 4))))
  (mattcheck "unique-pairs"
             (unique-pairs target)
             answer))
<<echo>>
<<prime-sum-txt>>
(define (prime-sum-pairs-mine n)
  (map make-pair-sum
       (filter prime-sum?
               (unique-pairs n))))
(let ((answer (list (list 2 1 3) (list 3 2 5)
                    (list 4 1 5) (list 4 3 7)
                    (list 5 2 7)) ))
  (mattcheck "prime-sum-pairs with unique-pairs"
             (prime-sum-pairs-mine 5)
             answer))
```

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

3.46 Exercise 2.41: Ordered triples of positive integers

ID: f29f74e9-e2a6-4af9-b885-4c5ec0b4253f

3.46.1 Question

ID: 5c6311c9-709f-4703-b1e4-a225f64e9312

> 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

ID: 865a33a4-b121-43e1-b16f-25de2c1d4897

```
<<fold-compat>>
<<enumerate-interval>>
<<flatmap>>
(define (unique-triplets n)
  (flatmap (lambda(i)
             (flatmap (lambda(j)
                        (map (lambda(k) (list i j k))
                             (enumerate-interval 1 (- j 1))))
                      (enumerate-interval 1 (- i 1))))
           (enumerate-interval 1 n)))
```

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

```
(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

ID: db842530-7a5d-4f0f-af80-0dd4c2e115f7

3.47.1 Question

ID: 67862610-e761-4def-9374-735130432089

The “eight-queens puzzle” asks how to place eight queens on a chessboard so that no queen is in check from any other.

```
<<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))
                 (enumerate-interval 1 board-size)))
          (queen-cols (- k 1))))))
  (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 k^{th} 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

ID: 3de2fa03-7508-4172-b9c3-aba6d64818ac

```
(define empty-board '())
(define (adjoin-position new-row column rest-of-queens)
  (cons (list new-row column) rest-of-queens))
(define (newer-position board)
  (car board))
(define (older-positions board)
  (cdr board))
```

```scheme
  (define (get-row position)
    (car position))
  (define (get-column position)
    (cadr position))
  (define (safe? k board)
    (define (same-row? a b)
      (= (get-row a) (get-row b)))
    (define (diagonal? a b)
      (let ((row-diff (abs (- (get-row a) (get-row b))))
            (col-diff (abs (- (get-column a) (get-column b)))))
        (= row-diff col-diff)))
    (let* ((new (newer-position board))
           (compare (older-positions board)))
      (and-map (lambda(pos)
                 (not (or (same-row? new pos)
                          (diagonal? new pos))))
               compare)))
```

```scheme
<<queens-txt>>
<<queens-mine>>
(load "mattcheck2.scm")
(let ((q4 '(((3 4) (1 3) (4 2) (2 1))
            ((2 4) (4 3) (1 2) (3 1))))
      (q11l 2680))
  (mattcheck "queens"
             (list (queens 4)
                   (length (queens 11)))
             (list q4 q11l)))
```

```
SUCCEED at queens
```

3.48 Exercise 2.43: Louis' `queens`

ID: 994c8401-9a1a-434e-8d78-3742c0c9d522

3.48.1 Question

ID: 787321bd-b16a-4c95-a19e-d42b2c8aa1fa

> Louis Reasoner is having a terrible time doing Exercise 2.42. His `queens` 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

```
(flatmap
 (lambda (new-row)
   (map (lambda (rest-of-queens)
          (adjoin-position new-row k rest-of-queens))
        (queen-cols (- k 1))))
 (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

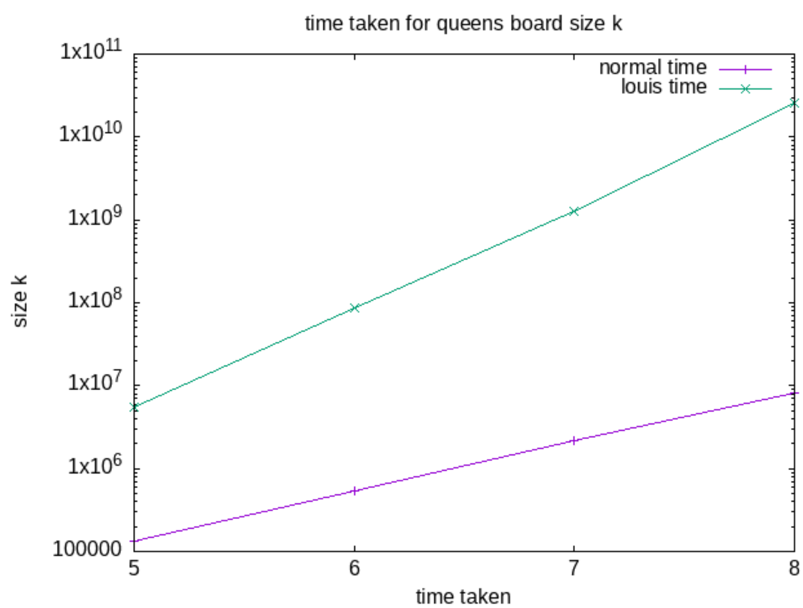
ID: ac70f63f-d952-469a-9127-2d943557b879

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.

```
<<queens-txt>>
<<queens-mine>>
(load "../mattbench.scm")

(define (queens-louis board-size)
  (define (queen-cols k)
    (if (= k 0)
        (list empty-board)
        (filter
         (lambda (positions) (safe? k positions))
         (flatmap
          (lambda (new-row)
            (map (lambda (rest-of-queens)
                   (adjoin-position new-row k rest-of-queens))
                 (queen-cols (- k 1))))
          (enumerate-interval 1 board-size)))))
  (queen-cols board-size))

(define (test size)
  (format #t "~&normal queens x~a: ~a" size
          (mattbench (lambda()(queens size)) 10000))
  (format #t "~&swapped queens x~a: ~a" size
          (mattbench (lambda()(queens-louis size)) 1000)))

(map (lambda(n)
```

```
        (test n))
      (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
```



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

ID: 8588fd65-1b81-43b8-855f-4fed731fb88f

[3] https://wernerdegroot.wordpress.com/2015/08/01/sicp-exercise-2-43/

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:

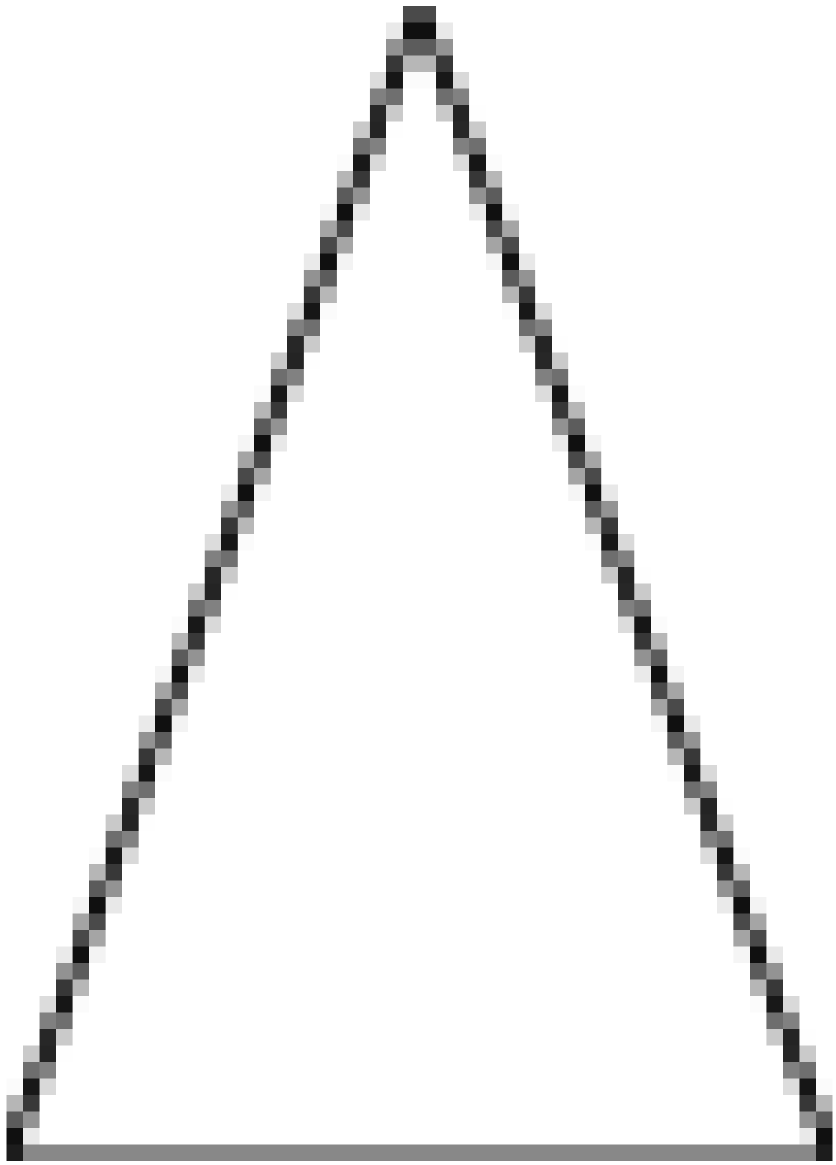
  - *beside* takes two painters, producing a new painter that puts one in the left half and one in the right half.

  - *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")
```

[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?

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

```
        0.5)
  ”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.

```
;; tilting at windmills trying to make a text-only picture language
;; TODO, probably won't finish
(define pl-defaultsize 32)
(define (make-painter orientation height width)
  (cons orientation (cons a b)))
(define (p-orientation p)
  (car p))
(define (p-height p)
  (cadr p))
(define (p-width p)
  (cddr p))
(define (opposite orientation)
  (cond ((eq? orientation 'down) 'up)
        ((eq? orientation 'up ) 'down)
        ((eq? orientation 'left) 'right)
        ((eq? orientation 'right) 'left)))
(define (flip-p p)
  (make-painter (opposite (p-orientation p))))
(define (below a b)
  (make-painter 'down a b))
(define (beside a b)
  (make-painter 'right a b))
(define (flip-vert p)
```

```
  (if (pair? p)
      (make-painter ))
(define (paint pict)
  (define (rect p x y)
    (cond ((string? p)
          )))
  (rec p pl-defaultsize pl-defaultsize))
(define (beside a b)
  (hc-append a b))
(define (below a b)
  (vc-append a b))
```

3.50 Exercise 2.44: `up-split`

ID: dde0ab90-49dd-4d62-9e6b-c636959c20c6

3.50.1 Text Definitions

ID: b0a64d21-e7fc-4ef6-a374-26741e1d5303

```
(define (right-split painter n)
  (if (= n 0)
      painter
      (let ((smaller (right-split painter
                                  (- n 1))))
        (beside painter
                (below smaller smaller)))))
```

3.50.2 Question

ID: d54717c6-0e99-42ef-a5db-a2405bb849bf

> 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

ID: 6bd6c390-e052-420a-9b0e-67389c884b1b

```
(define (up-split painter n)
  (if (= n 0)
      painter
      (let ((smaller (up-split painter
                               (- n 1))))
        (below painter
               (beside smaller smaller)))))
```

3.51 Exercise 2.45: Generalized splitting

ID: 1450bd14-52fd-45d6-b249-868093976032

3.51.1 Question

ID: 3f151d8e-b88e-42cd-a812-c87e13529dd9

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

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

> produces procedures right-split and up-split with the same behaviors as the ones already defined.

3.51.2 Answer

ID: cf4b601e-6eb1-4c46-a2b6-c89279497b13

```
(define (split f1 f2)
  (define (rec painter n)
    (if (= n 0)
        painter
        (let ((smaller (rec painter
                            (- n 1))))
          (f1 painter
              (f2 smaller smaller)))))
  rec)
(define right-split (split beside below))
;; This one goes down for some reason?
;(define up-split (split below beside))
(define (up-split painter n)
  (if (= n 0)
      painter
      (let ((smaller (up-split painter
                               (- n 1))))
        (below painter
               (beside smaller smaller)))))
```

3.52 Exercise 2.46: Defining vectors

ID: cfdbca96-f763-4b61-9823-bd3363e0b385

3.52.1 Question

`ID: eb5879a3-09a1-4819-9fc7-c2acd2c0f9e2`

> 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:
>
> $$\begin{aligned}(x_1, y_1) + (x_2, y_2) &= (x_1 + x_2, y_1 + y_2),\\ (x_1, y_1) - (x_2, y_2) &= (x_1 - x_2, y_1 - y_2),\\ s \cdot (x, y) &= (sx, sy).\end{aligned}$$

3.52.2 Answer

`ID: 75604738-82fe-47e9-ac83-889e2fa8bd7d`

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))))
```

```scheme
(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

ID: eedeb3cf-97f6-4e91-8914-6308ad182504

3.53.1 Question

ID: 0017b75c-f2c6-411e-94d7-8bdb8c2de92c

> 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.

3.53.2 Answer

ID: 2465aa1f-271c-4a42-9b82-d9d838efc061

```scheme
(define (make-frame origin edge1 edge2)
  (list origin edge1 edge2))
(define (origin-frame F)
  (car F))
(define (edge1-frame F)
  (cadr F))
(define (edge2-frame F)
  (caddr F))
```

```scheme
(define (make-frame origin edge1 edge2)
  (cons origin (cons edge1 edge2)))
(define (origin-frame F)
  (car F))
(define (edge1-frame F)
  (cadr F))
```

```scheme
(define (edge2-frame F)
  (cddr F))
```

3.54 Exercise 2.48: Line segments

ID: d4e49e3e-ec42-470a-a1ca-903715953a43

3.54.1 Question

ID: d1775142-e3ba-4227-a2e9-950de381ae0c

> 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-segment` and selectors `start-segment` and `end-segment`.

3.54.2 Answer

ID: 814704f8-5858-4324-ac28-c9b43ed7e86f

Again reminding me of Exercise 2.2.

```scheme
(define (make-segment start end)
  (cons start end))
(define (start-segment seg)
  (car seg))
(define (end-segment seg)
  (cdr seg))
```

3.55 Exercise 2.49: Primitive painters

ID: 7913c8e9-10fa-4d5d-aafc-2c87f83e8ab2

3.55.1 Text Definitions

ID: 97444370-d19f-4b80-a39f-d707e8721801

```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))))))
```

```scheme
(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

`ID: 32e57911-7ed3-4844-9231-578c641a3afb`

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

`ID: be7dd3d7-02fd-437e-9476-d5eb0112c4a7`

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)))))
(define frame-X
  (segments->painter (list
   (make-segment (make-vect 0 0)
                 (make-vect 1 1))
   (make-segment (make-vect 0 1)
                 (make-vect 1 0)))))
```

```
(define diamond
  (segments->painter (list
   (make-segment (make-vect 0.5 0)
                 (make-vect 1 0.5))
   (make-segment (make-vect 1 0.5)
                 (make-vect 0.5 1))
   (make-segment (make-vect 0.5 1)
                 (make-vect 0 0.5))
   (make-segment (make-vect 0 0.5)
                 (make-vect 0.5 0)))))
```

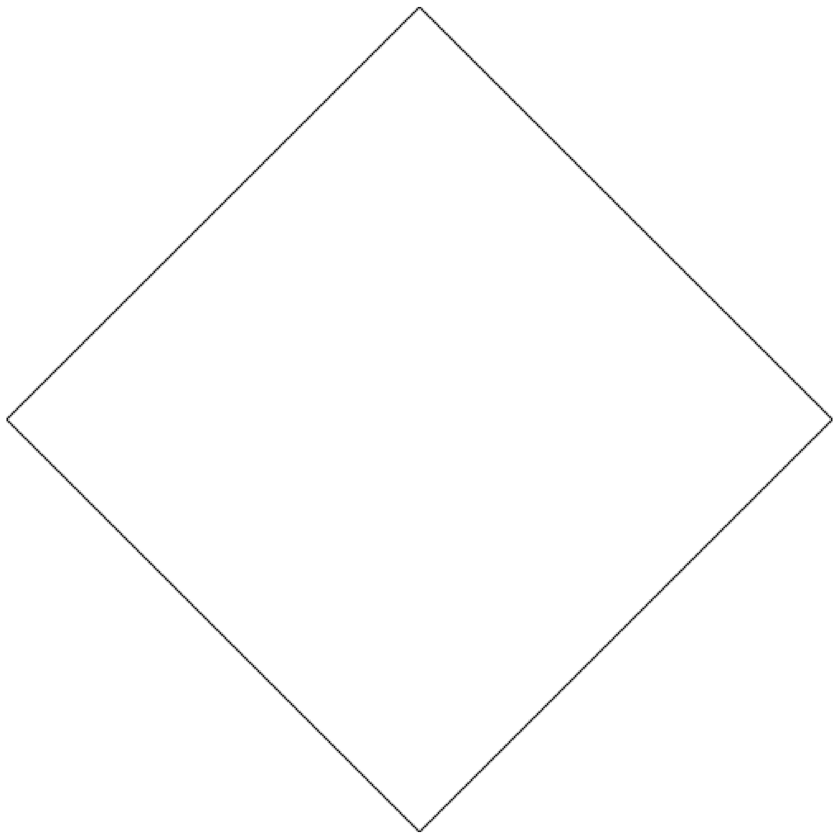
I'll make the wave painter once I have some graph paper in front of me.

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.

```
(use-modules (pict))
<<make-frame>>
<<make-vect>>
<<make-segment>>
(define (frame-coord-map frame)
  ;; Returns a function for adjusting a frame by a vector
  (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 (draw-line start end)
  ;; take two vectors, returns a line SVG object for pict
  (line (xcor-vect start)
        (ycor-vect start)
        (xcor-vect end)
        (ycor-vect end)))
(define (segments->painter segment-list)
  ;; takes a list of segments, returns a "painter" lambda, which applies a frame
  ;; to those segments and then maps over the result with draw-line to make a
  ;; list of SVG line objects which pict can combine.
  (lambda (frame)
    (map
     (lambda (segment)
       (draw-line
        ((frame-coord-map frame)
         (start-segment segment))
```

```
        ((frame-coord-map frame)
         (end-segment segment))))
     segment-list)))
;; NOTE: in the text, draw-line is a function which triggers an action in
;; some graphics driver, and returns nothing. Because of this, (map) was
;; originally (for-each). Thus the final result would have been thrown away.

<<painters>>

(define (paint-lines painter)
  ;; use pict to compile an SVG with the elements described by painter
  (let ((Frame (make-frame (make-vect 0 0)
                           (make-vect 500 0)
                           (make-vect 0 500))))
    (apply lt-superimpose
           (painter Frame))))
```

```
<<linepainter-pict>>
(pict->file (paint-lines diamond)
            "2/pict/testline.svg")
; #:maxw 100 #:maxh 100) <- FIXME: This procedure should take these arguments
; but I can't get it to work.
```



Holy moly it actually works.

3.56 Exercise 2.50: Transforming painters

ID: 028d6bfc-468d-4ab1-86d3-673f1a1e4e43

3.56.1 Text Definitions

ID: a6debaa2-a28d-42bd-bccc-1dc85cfc6841

```
(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
   painter
   (make-vect 0.0 1.0)   ; new origin
   (make-vect 1.0 1.0)   ; new end of edge1
   (make-vect 0.0 0.0))) ; new end of edge2

(define (rotate90 painter)
  (transform-painter painter
                     (make-vect 1.0 0.0)
                     (make-vect 1.0 1.0)
                     (make-vect 0.0 0.0)))

(define (squash-inwards painter)
  (transform-painter painter
                     (make-vect 0.0 0.0)
                     (make-vect 0.65 0.35)
                     (make-vect 0.35 0.65)))
(define (beside painter1 painter2)
  (let ((split-point (make-vect 0.5 0.0)))
    (let ((paint-left  (transform-painter
                        painter1
                        (make-vect 0.0 0.0)
                        split-point
                        (make-vect 0.0 1.0)))
          (paint-right (transform-painter
                        painter2
                        split-point
                        (make-vect 1.0 0.0)
                        (make-vect 0.5 1.0))))
      (lambda (frame)
        (append
         (paint-left frame)
         (paint-right frame))))))
```

```
<<linepainter-pict>>
<<frame-transforms-txt>>
(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 0.4 0))
                      (make-segment (make-vect 0.4 0)
                                    (make-vect 0 0)))))
(let ((picture (beside (beside topleft-tri
```

```
                                   topleft-tri)
                           (rotate90 topleft-tri))))
(pict->file (paint-lines picture)
            "2/pict/rotatetest.svg"))
```

3.56.2 Question

ID: 1f5a11df-2b2e-49e9-b301-3c84a857aced

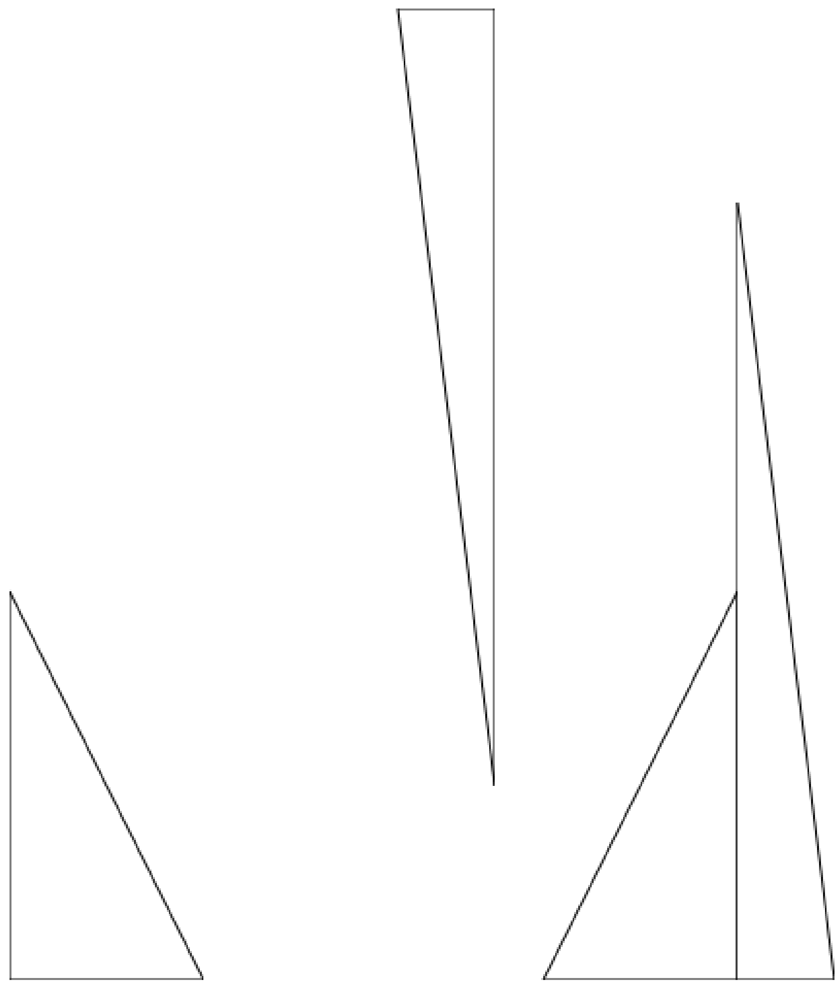
> Define the transformation flip-horiz, which flips painters horizontally, and transformations that rotate painters counterclockwise by 180 degrees and 270 degrees.

3.56.3 Answer

ID: 985538a4-a01a-45c1-b025-68c7845c55c8

```
(define (flip-horiz painter)
  (transform-painter
   painter
   (make-vect 1.0 0.0)   ; new origin
   (make-vect 0.0 0.0)   ; new end of edge1
   (make-vect 1.0 1.0))) ; new end of edge2

(define (rotate180 painter)
  (transform-painter painter
                     (make-vect 1.0 1.0)
                     (make-vect 0.0 1.0)
                     (make-vect 1.0 0.0)))
(define (rotate270 painter)
  (transform-painter painter
                     (make-vect 0.0 1.0)
                     (make-vect 0.0 0.0)
                     (make-vect 1.0 1.0)))
```

```
<<linepainter-pict>>
<<frame-transforms-txt>>
<<flip-rotate2>>
(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 ((picture (beside (beside
                         (flip-vert topleft-tri)
                         (rotate90 topleft-tri))
                        (beside
                         (rotate180 topleft-tri)
                         (rotate270 topleft-tri)))))
  (pict->file (paint-lines picture)
              "2/pict/rotate2.svg"))
```



3.57 Exercise 2.51

ID: 657b9460-c0c5-4f37-9dcc-8310d828d448

3.57.1 Question

ID: fff4a3e4-e368-4acd-aa67-5d689830761a

> 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}).

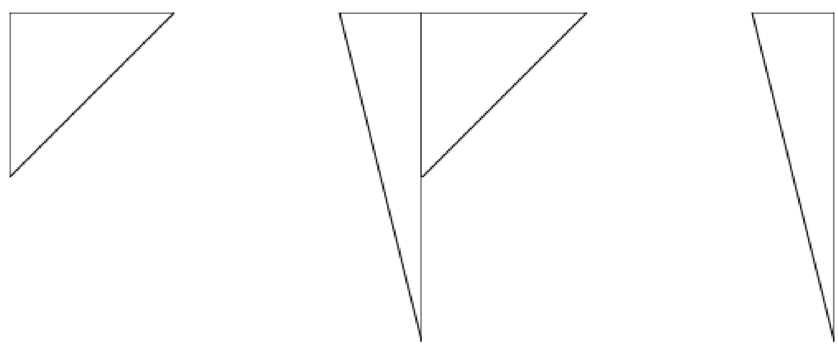
3.57.2 Answer

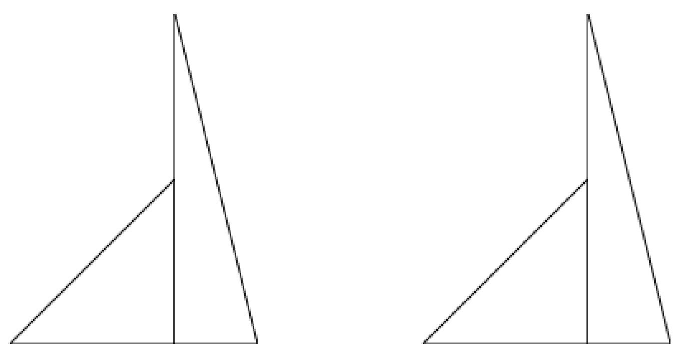
ID: 7ceba648-533f-4a5c-9368-37de4e9cc8bf

```
(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
                  topleft-tri
                  (rotate90 topleft-tri))
                 (beside
                  (rotate180 topleft-tri)
                  (rotate270 topleft-tri))))
      (p2 (below-rotate (beside
                  topleft-tri
                  (rotate90 topleft-tri))
                 (beside
                  (rotate180 topleft-tri)
```

```
                        (rotate270 topleft-tri)))))
  (pict->file (paint-lines (beside p1 p2))
              "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

`ID: 35ea15ae-3459-4f78-999e-5d61928c5eb5`

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

ID: fcc09b41-5e26-483c-9983-109499a6b031

3.59.1 Question A

ID: 45bbf4fd-704e-4124-a54b-34012f42f5a9

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

ID: 7e78cf22-b210-4018-9cc8-a88c3e1f7abb

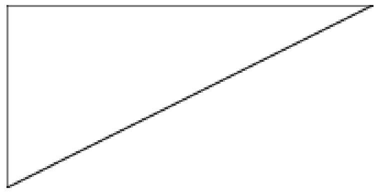
I need to do what I've been slacking off on: actually making the `wave` painter. First, I want a helper function to make inputting shapes easier.

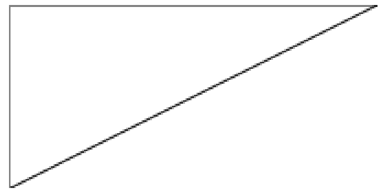
```
(define (unwrap-if-needed args)
  ;; commonly needed in vararg functions
  (if (and (= 1 (length args))
           (list? (car args)))
      (car args) ; assume we were passed a list, unwrap it
      args))
```

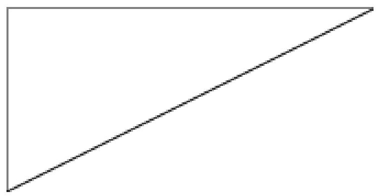
```
<<unwrap-if-needed>>
(define (continuous-lines . vectors)
  ;; Given a list of vectors, return a series of line segments
  ;; that continuously follow the vectors until there are no more,
  ;; at which point connect the last vector to the first.
  (define (iter vecs lines)
    (if (> 2 (length vecs))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

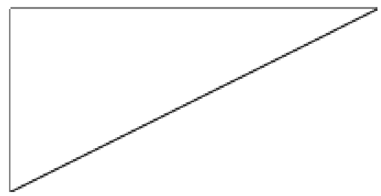
```
          (iter rest (cons new-line lines)))))
  (let ((input (unwrap-if-needed vectors)))
    (iter (append input (list (car input))) ;; Connect to start
          '())))
(define (numbers-to-vectors . numbers)
  (define (iter input output)
    (if (> 2 (length input))
        output
        (let* ((first (car input))
               (d (cdr input))
               (second (car d))
               (dd (cdr d)))
          (iter dd
                (cons (make-vect first second)
                      output)))))
  (reverse (iter numbers '())))
```

```
<<linepainter-pict>>
<<frame-transforms-txt>>
<<flip-rotate2>>
<<below>>
<<continuous-lines>>
(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)))))
(define topleft-tri2
  (segments->painter (continuous-lines
                      (make-vect 0 0)
                      (make-vect 0 0.4)
                      (make-vect 0.8 0))))
(define topleft-tri3
  (segments->painter (continuous-lines
                      (numbers-to-vectors
                       0 0
                       0 0.4
                       0.8 0))))
(let ((p1 (below (beside topleft-tri topleft-tri2)
                 (beside topleft-tri topleft-tri3))))
  (pict->file (paint-lines p1)
              "2/pict/cl-test.svg"))
```









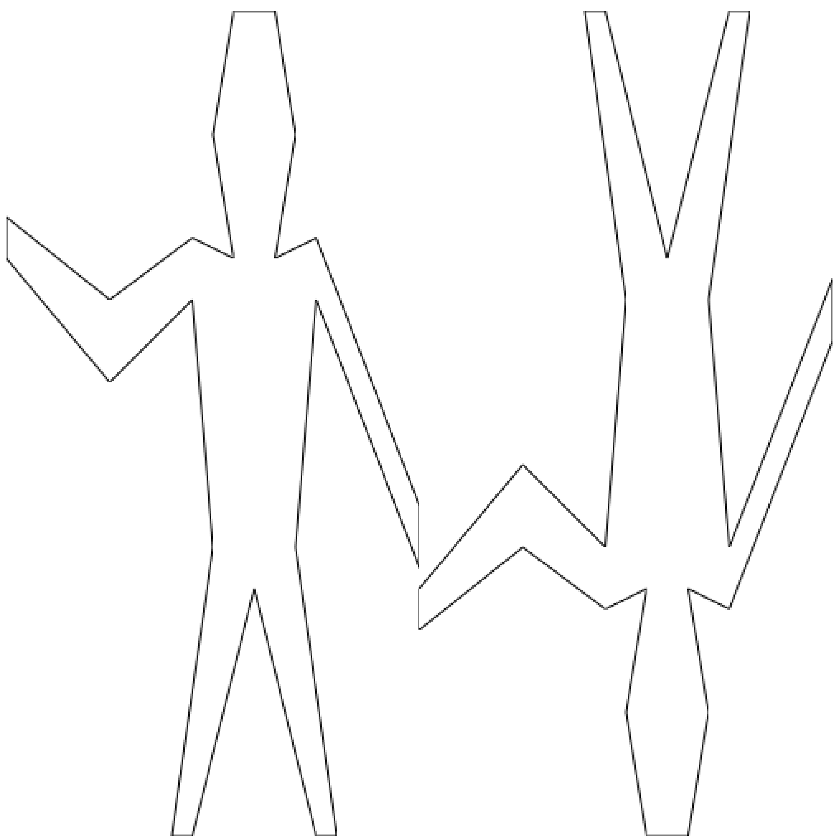
Now, let's write the wave painter.

```
<<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
```

```
      1 0.675
      1 0.6
      0.75 0.275
      0.65 0.3
      0.7 0.15 ;; center-right side of head
      0.65 0))))
```

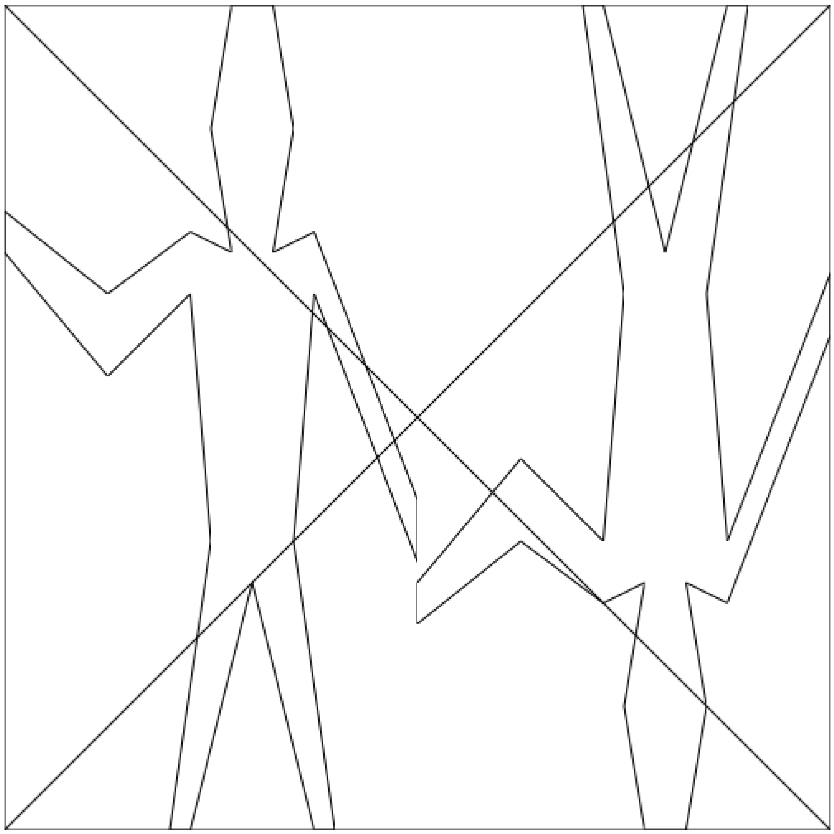
```
<<linepainter-pict>>
<<frame-transforms-txt>>
<<flip-rotate2>>
<<below>>
<<continuous-lines>>
<<wave-painter>>
(let ((p1 (beside wave (flip-vert wave))))
  (pict->file (paint-lines p1)
              "2/pict/wave-test.svg"))
```

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.



```
<<unwrap-if-needed>>
(define (append-painters-rec . args)
  (lambda (frame)
    (define (rec painters)
      (if (null? painters)
          '()
          (append ((car painters) frame)
                  (rec (cdr painters)))))
    (rec (unwrap-if-needed args))))
(define (append-painters . args)
  (lambda (frame)
    (define (iter segments to-paint)
      (if (null? to-paint)
          segments
          (iter (append ((car to-paint) frame)
                        segments)
                (cdr to-paint))))
    (iter '()
          (unwrap-if-needed args))))
```

```
<<linepainter-pict>>
<<frame-transforms-txt>>
<<flip-rotate2>>
<<below>>
<<continuous-lines>>
<<painters>>
<<wave-painter>>
<<append-painters>>
(let ((p1 (append-painters
           (beside wave (flip-vert wave))
           frame-X
           outline)))
  (pict->file (paint-lines p1)
              "2/pict/append-test.svg"))
```

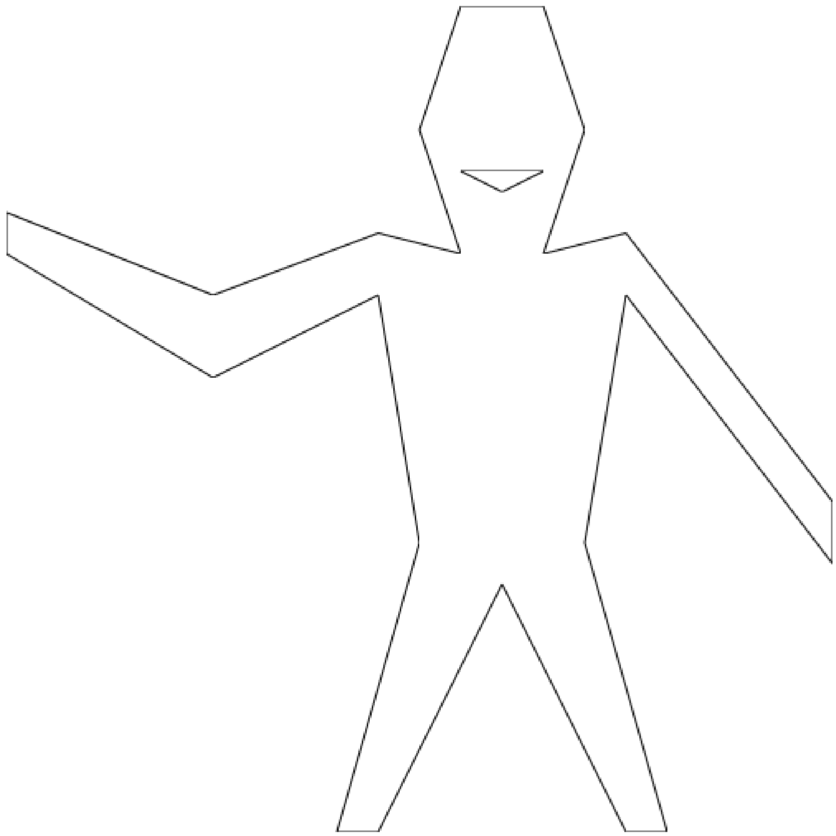


Now let's (try to) add a smile.

```
<<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>>
```

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



3.59.3 Question B

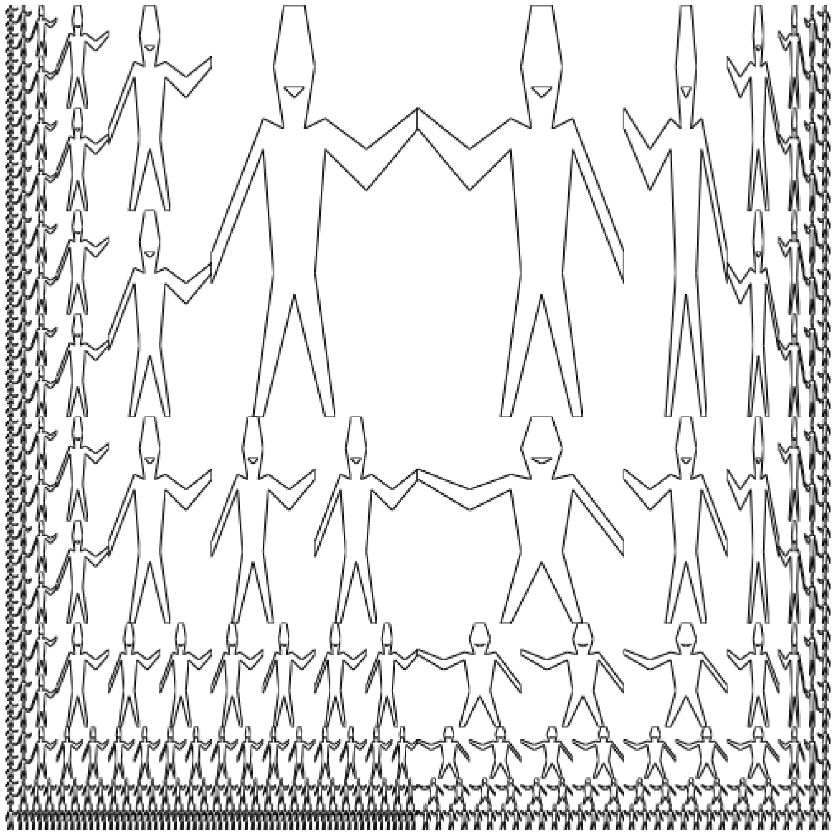
ID: 129b6c3e-257f-4e9a-bd96-e8e13f4d5cc5

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).

```
(define (corner-split painter n)
  (if (= n 0)
      painter
      (let ((up (up-split painter (- n 1)))
            (right (right-split painter
                                (- n 1))))
        (let ((top-left (beside up up))
              (bottom-right (below right
```

```
                                     right))
                 (corner (corner-split painter
                                      (- n 1))))
             (beside (below painter top-left)
                     (below bottom-right
                            corner))))))

(define (corner-split-mine painter n)
  (if (= n 0)
      painter
      (let ((up (up-split painter (- n 1)))
            (right (right-split painter
                                (- n 1))))
        (let ((top-left up)
              (bottom-right right)
              (corner (corner-split-mine painter
                                      (- n 1))))
          (beside (below painter top-left)
                  (below bottom-right
                         corner))))))
```

```
<<linepainter-pict>>
<<frame-transforms-txt>>
<<flip-rotate2>>
<<below>>
<<continuous-lines>>
<<painters>>
<<wave-smile>>
<<append-painters>>
<<splits-generalized>>
<<corner-split>>
(let ((p1 (beside (flip-horiz (corner-split wave-smile 5))
                  (corner-split-mine wave-smile 5))))
;(let ((p1 (up-split frame-X 5)))
  (pict->file (paint-lines p1)
              "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

ID: 3c3aea71-197f-44cc-8ed0-97b464a037ec

> 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

ID: b0a6d67f-6886-4a55-92fe-b233944c206a

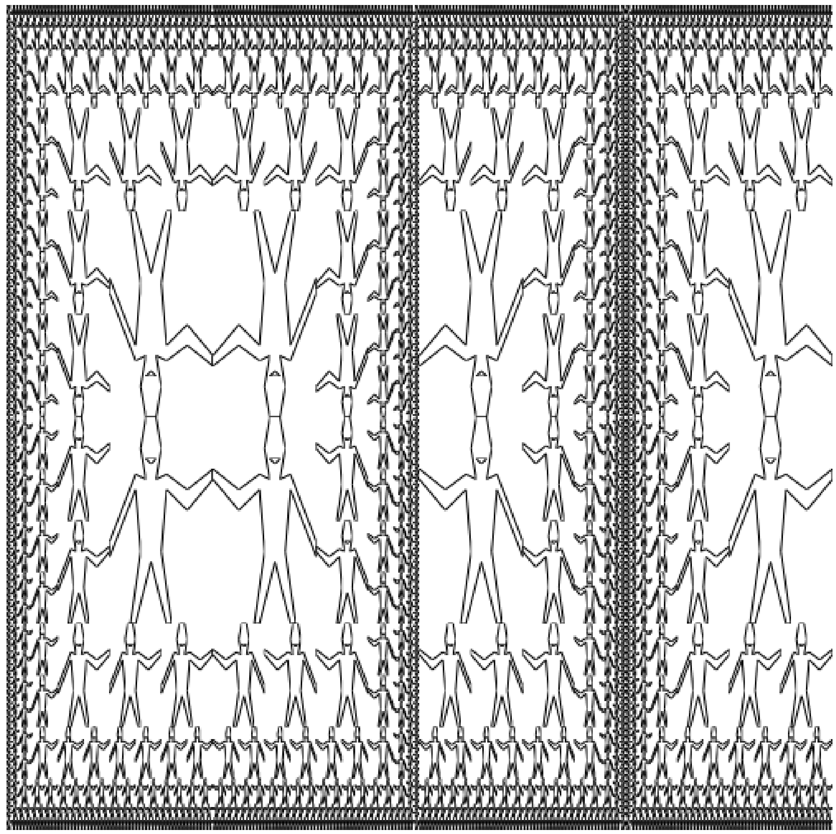
```
(define (square-of-four tl tr bl br)
  (lambda (painter)
    (let ((top (beside (tl painter)
                       (tr painter)))
          (bottom (beside (bl painter)
                          (br painter))))
      (below bottom top))))
(define (square-limit painter n)
  (let ((combine4
         (square-of-four flip-horiz
                         identity
                         rotate180
                         flip-vert)))
    (combine4 (corner-split painter n))))
```

3.59.6 Answer C

ID: 91c819a1-d908-4957-9713-ab4c6d282166

```
(define (square-limit-mine painter n)
  (let ((combine4
         (square-of-four identity
                         flip-horiz
                         flip-vert
                         rotate180)))
    (combine4 (corner-split painter n))))
```

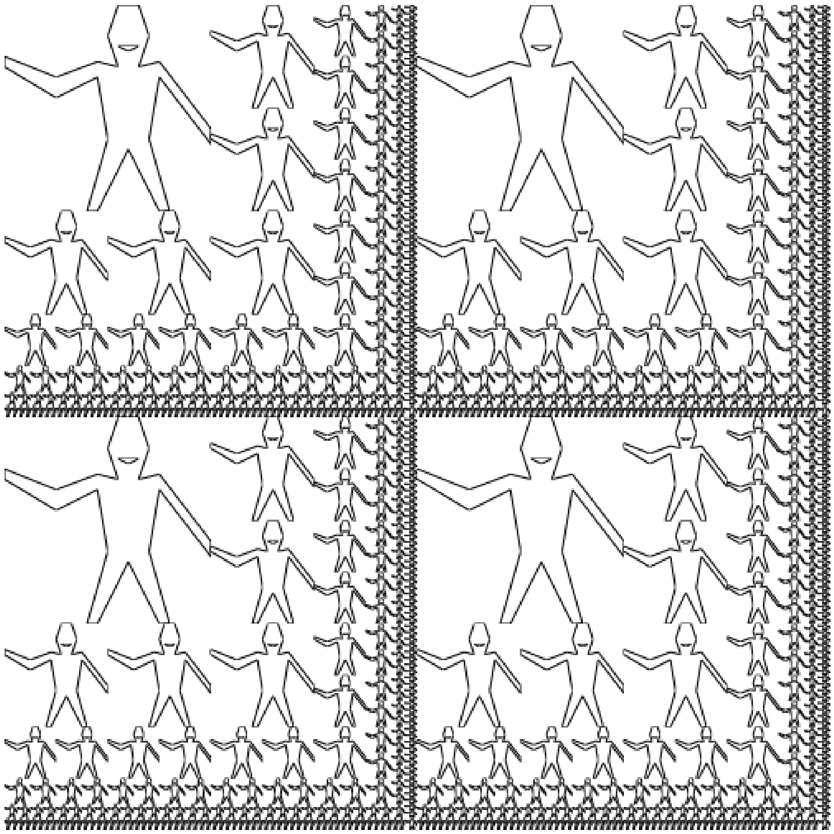
```
<<linepainter-pict>>
<<frame-transforms-txt>>
<<flip-rotate2>>
<<below>>
<<continuous-lines>>
<<wave-smile>>
<<splits-generalized>>
<<corner-split>>
<<square-of-four-txt>>
<<square-limit-mine>>
(pict->file (paint-lines (beside (square-limit wave-smile 5)
                                 (square-limit-mine wave-smile 5)))
            "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.

```
<<linepainter-pict>>
<<frame-transforms-txt>>
<<flip-rotate2>>
<<below>>
<<continuous-lines>>
<<wave-smile>>
<<splits-generalized>>
<<corner-split>>
<<square-of-four-txt>>
(define (square-limit-mine painter n)
  (let ((combine4
         (square-of-four identity
                         identity
                         identity
                         identity)))
```

```
        (combine4 (corner-split painter n))))
  (pict->file (paint-lines (square-limit-mine wave-smile 5))
              "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

ID: f1cb5c0f-0137-4642-878c-242ff44b91cb

Syntactic vs semantic use in Lisp:

```
(define dog 123)
```

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

```
(+ dog dog) ;; semantic usage
(quote dog) ;; syntactic usage
```

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

3.61 Exercise 2.53

ID: 258696d9-05d2-49ec-b35e-3edb680427a2

3.61.1 Question

ID: e0034d4c-fbfe-4a87-ae13-ed845db2b81e

> 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

ID: 582a441c-1a57-40e9-bb8d-c382dabbaa86

```
(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

ID: 29527ad8-cc9f-4d2e-9d8a-6475e783dc85

> 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

ID: 3043a43e-7468-4b14-96e8-49490e3056f2

```
(define (equal? a b)
  (cond ((and (symbol? a)
              (symbol? b))
         (eq? a b))
        ((and (list? a)
              (list? b))
         (and (equal? (car a) (car b))
              (equal? (cdr a) (cdr b))))
        (else #f)))
```

```
(load "mattcheck2.scm")
(let ((a '(this is a list))
      (aa '(this is a list))
      (b '(this (quote is a) list)))
  (mattcheck "equal? true"
             (equal? a a))
  (mattcheck "equal? trick question"
             (equal? a a))
  (mattcheck "equal? false"
             (equal? a b)
             #f))
```

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

3.63 Exercise 2.55

ID: 00be831b-8f4c-425f-bcb1-7ad613ddb06f

3.63.1 Question

ID: 87cfa7f0-088a-4d47-9203-bb587ca4d503

Eva Lu Ator types to the interpreter the expression

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

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

3.63.2 Answer

`ID: 567cc140-30bf-40bf-8fff-889862d517cb`

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

3.64 2.3.2: Example: Symbolic differentiator

`ID: 1f159d89-0a02-4726-9872-bbbc527ab559`

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:

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

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

ID: 6be5f14d-c225-46d2-a1d4-1864083a8c76

3.65.1 Text definitions

ID: b0b61dc7-4677-4b9f-8249-edf8beb99189

```
(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))))
```

```scheme
(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

ID: c2acf981-29fc-4cf7-9afd-56f7764b11d8

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

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

by adding a new clause to the `deriv` program and defining appropriate procedures `exponentiation?`, `base`, `exponent`, and `make-exponentiation`. (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

ID: 5943cb2c-8dd3-4789-ab52-9d74eee215aa

```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.

```
<<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)
           'y)
(mattcheck ”deriv: bad simplification”
           (deriv '(* (* x y) (+ x 3)) 'x)
           '(+ (* x y) (* y (+ x 3))))
;; Now, did I do my job right?
;; Checking against this guy's results:
;;   http://jots-jottings.blogspot.com/2011/11/sicp-exercise-256-differe⌟
↪  ntiating.html
(mattcheck ”make-exponentiation 1”
           (make-exponentiation 1 12)
           1)
(mattcheck ”make-exponentiation 2”
           (make-exponentiation 2 12)
           4096)
(mattcheck ”make-exponentiation 3”
           (make-exponentiation 'x 12)
           '(** x 12))
(mattcheck ”deriv: exponentiation 1”
           (deriv (make-exponentiation 'a 5) 'a)
           '(* 5 (** a 4)))
(mattcheck ”deriv: exponentiation 2”
           (deriv (make-exponentiation 'a 'b) 'a)
```

```
            '(* 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

ID: 47f5c525-98f5-403f-9dd1-df43fd4b5155

3.66.1 Question

ID: a73afb4a-3e3b-4d8a-b908-3e9f2bc051f4

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

```
(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

ID: e970b43b-d753-43e1-b844-7ee3e342a2dc

```
(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))))
```

```
<<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:

```
(define (augend s)
  (let ((rest (cddr s)))
    (if (null? (cdr rest))
        (car rest)
;        (make-sum (addend rest)
;                  (augend (cons '+ rest))))))
        (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

ID: 0c266a88-a666-42f8-8da6-758186f84570

3.67.1 Question

ID: 435005e1-fd26-4326-a1ab-827d7f6e9ca8

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

ID: e9680d8c-53f0-4d51-a89e-0ceedb068100

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

ID: b54640f3-881c-46d2-977d-ee14cc6f8664

```scheme
(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) (caddr s))
(define (product? x) (and (pair? x) (eq? (cadr x) '*)))
(define (multiplier p) (car 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))))
```

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

ID: e13be4f0-5c38-4da2-9c98-453a90730e1d

The problem becomes substantially harder if we allow standard algebraic notation, 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

ID: 2e7a512e-72a1-4728-bc06-e6a140c70867

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))))
(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))))
```

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

;; Test cases from:
;; http://community.schemewiki.org/?sicp-ex-2.58
(mattcheck "deriv: mult prioritization 1"
           (deriv '(x + 3 * (x + y + 2)) 'x)
           4)
(mattcheck "deriv: mult prioritization 2"
           (deriv '(x * (y * (x + 3))) 'x)
           '((x * y) + (y * (x + 3))))
(mattcheck "deriv: mult prioritization 3"
           (deriv '((x * y) * (x + 3)) 'x)
           '((x * y) + (y * (x + 3))))
;; Checks from "future Matt"
(mattcheck "deriv: mult prioritization 4a"
           (deriv '(x + 3 + x + y * 2) 'x)
           2)
(mattcheck "deriv: mult prioritization 4b"
           (deriv '(y * 2 + x + 3 + x) 'x)
```

```scheme
            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:

```scheme
;; I want to do f to the first item of x, unless some-condition? in the second
;; item, in which case do g to the 2nd item before x to the first
(define (foo x)
  (if (some-condition? (cdr x))
      (f (cons (car x)
               (g (cdr x))))
      (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:

```scheme
(define func-list '(product? sum?))
(define priority-list '(* +))
(define (solve-in-order exp func-list)
  (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)
                        (deriv (multiplicand exp) var))
```

```scheme
           (make-product (deriv (multiplier exp) var)
                         (multiplicand exp))))
         (else
          (error "unknown expression type: DERIV" exp))))
```

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

;; Test cases from:
;; http://community.schemewiki.org/?sicp-ex-2.58
(mattcheck "deriv: mult prioritization 1"
           (deriv '(x + 3 * (x + y + 2)) 'x)
           4)
(mattcheck "deriv: mult prioritization 2"
           (deriv '(x * (y * (x + 3))) 'x)
           '((x * y) + (y * (x + 3))))
(mattcheck "deriv: mult prioritization 3"
           (deriv '((x * y) * (x + 3)) 'x)
           '((x * y) + (y * (x + 3))))
;; Checks from "future Matt"
(mattcheck "deriv: mult prioritization 4a"
           (deriv '(x + 3 + x + y * 2) 'x)
           2)
(mattcheck "deriv: mult prioritization 4b"
           (deriv '(y * 2 + x + 3 + x) 'x)
           2)
```

3.68 Exercise 2.59: Representing sets

ID: ecbb093d-99c4-44c1-9891-ab68c5dd3c31

3.68.1 Text definitions

ID: 9e32f631-f5d6-4528-96fb-9899c3ad835b

```
(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

ID: e4853186-1af6-495c-96f7-008add26ce22

Implement the `union-set` operation for the unordered-list representation of sets.

3.68.3 Answer

ID: 8d7e0c34-2dd5-40f9-ac98-4b3abb8c4c58

`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.

```
(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)
                   (cons (car s)
                         result)))))
    (append (iter set1 '())
            set2))
```

```
<<sets-txt>>
<<union-set>>
(load "mattcheck2.scm")
(let ((set1 (list 1 2 3 4 5 6))
      (set2 (list 4 5 6 7 8 9))
      (set-union (list 1 2 3 4 5 6 7 8 9)))
  (mattcheck "union-set-rec"
             (union-set-rec set1 set2)
             set-union)
  (mattcheck "union-set"
             (union-set set1 set2)
             set-union))
```

```
SUCCEED at union-set-rec
SUCCEED at union-set
```

```
<<enumerate-interval>>
<<sets-txt>>
<<union-set>>
(load "../mattbench.scm")

(let ((set1 (enumerate-interval 1 1000))
      (set2 (enumerate-interval 500 1500)))
  (define (test)
    (format #t "~&recursive union-sets: ~a"
            (cadr (mattbench2 (lambda()(union-set-rec set1 set2))
            ↪  10000)))
    (format #t "~&iterative union-sets: ~a"
            (cadr (mattbench2 (lambda()(union-set set1 set2)) 10000))))

  (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

ID: 56d38582-cfa6-474e-acf8-7788a2e54b9f

3.69.1 Question

ID: 77e25c2d-043a-4ff1-a60e-5201a853288e

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-set`, `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

ID: 49bfc1b6-e8c0-4833-9be1-e535e23b5e1b

`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?

```scheme
(define (adjoin-set-dupes x set)
  (cons x set))

(define (union-set-dupes set1 set2)
  (append set1 set2))

(define (intersection-set-dupes set1 set2)
  (let ((inter (intersection-set set1 set2)) ;; yes, we're calling the non-duplicate version
        (union (union-set-dupes set1 set2)))
    (filter (lambda(x) (element-of-set? x inter))
            union)))
(define (intersection-set-dupes2 set1 set2)
  (let ((inter (intersection-set set1 set2)) ;; yes, we're calling the non-duplicate version
        (inter2 (intersection-set set2 set1)))
    (append inter inter2)))
```

```
<<sets-txt>>
<<union-set>>
<<set-dupes>>
(load ”mattcheck2.scm”)
(let ((set1 (list 1 2 3 4 5 6))
      (set2 (list 4 5 6 7 8 9))
      (set-union (list 1 2 3 4 5 6 7 8 9))
      (set-union-dupes
       (list 1 2 3 4 5 6 4 5 6 7 8 9))
      (set-intersection (list 4 5 6))
      (set-intersection-dupes
       (list 4 5 6 4 5 6)))
  (mattcheck ”union-set-dupes”
             (union-set-dupes set1 set2)
             set-union-dupes)
  (mattcheck ”union-set”
             (union-set set1 set2)
             set-union)
  (mattcheck ”intersection-set”
             (intersection-set set1 set2)
             set-intersection)
  (mattcheck ”intersection-set-dupes”
             (intersection-set-dupes set1 set2)
             set-intersection-dupes)
  (mattcheck ”intersection-set-dupes2”
             (intersection-set-dupes2 set1 set2)
             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
```

```
<<enumerate-interval>>
<<sets-txt>>
<<union-set>>
<<set-dupes>>
(use-modules (ice-9 format))
(load ”../mattbench.scm”)

(let ((set1 (enumerate-interval 1 1000))
      (set2 (enumerate-interval 500 1500)))
  (define (test)
    (format #t ”~&union-sets: ~a”
            (cadr (mattbench2 (lambda()(union-set set1 set2)) 10000)))
    (format #t ”~&union-sets-dupes: ~a”
```

```
            (cadr (mattbench2 (lambda()(union-set-dupes set1 set2))
            ↪  10000)))
    (format #t "~&intersection-set: ~a"
            (cadr (mattbench2 (lambda()(intersection-set set1 set2))
            ↪  10000)))
    (format #t "~&intersection-set-dupes: ~a"
            (cadr (mattbench2 (lambda()(intersection-set-dupes set1
            ↪  set2)) 10000)))
    (format #t "~&intersection-set-dupes2: ~a"
            (cadr (mattbench2 (lambda()(intersection-set-dupes2 set1
            ↪  set2)) 10000)))
    )

  (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

ID: 63d00424-b2ae-4c03-ba2d-7211587d0d58

3.70.1 Question

ID: 7835acdf-8bb9-400f-8e74-f74f17a5269c

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

ID: af210971-9110-42e6-a672-daed9e8c92a6

```
(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))
```

```
<<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
```

3.71 Exercise 2.62: union-set ordered

ID: 0f2a00ea-9015-4740-bd3f-dbd5d698b918

3.71.1 Question

ID: 12c67368-0159-4d19-856e-8f4f9f35f3a3

Give a $\Theta(n)$ implementation of `union-set` for sets represented as ordered lists.

3.71.2 Answer

ID: 8bab316a-a45a-4005-8d5d-330b4d1ea2e2

```
(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 '()))
```

```
<<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
```

```
<<enumerate-interval>>
<<sets-txt>>
<<union-set>>
<<union-set-ordered>>
(use-modules (ice-9 format))
(load "../mattbench.scm")

;; 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

ID: ab85e21b-827b-48eb-9ad2-58f6315bfddb

3.72.1 Text definitions

ID: 89e7156c-1d4d-4407-8f5c-376c8612f1fa

```
(define (entry tree) (car tree))
(define (left-branch tree) (cadr tree))
(define (right-branch tree) (caddr tree))
```

```scheme
(define (make-tree entry left right)
  (list entry left right))
```

3.72.2 Question A

ID: 175a45f6-69ab-46d4-981b-61dc99016c1e

Each of the following two procedures converts a binary tree to a list.

```scheme
<<make-tree>>
(define (tree->list-1 tree)
  (if (null? tree)
      '()
      (append (tree->list-1 (left-branch tree))
              (cons (entry tree)
                    (tree->list-1
                      (right-branch tree))))))
(define (tree->list-2 tree)
  (define (copy-to-list tree result-list)
    (if (null? tree)
        result-list
        (copy-to-list (left-branch tree)
                      (cons (entry tree)
                            (copy-to-list
                              (right-branch tree)
                              result-list)))))
  (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

ID: 49b808cb-71b0-4866-b1d7-1dda7ab4495a

First let's check whether the arrangement of the input tree impacts the output list. I'd like a couple functions for generating lists.

```scheme
(define (rightward-tree list)
  (if (null? list)
      '()
      (make-tree (car list) '()
                 (rightward-tree (cdr list)))))
(define (leftward-tree list)
  (define (rec list)
    (if (null? list)
        '()
        (make-tree (car list) (rec (cdr list))
```

```
                   '()))))
  (rec list))
(define (nested-tree list)
  (define (rec list)
    (cond ((= 1 (length list))
           (make-tree (car list) '() '()))
          ((= 2 (length list))
           (make-tree (cadr list)
                      (make-tree (car list) '() '())
                      '()))
          (else
           (let ((halfway (truncate/ (length list)
                                     2)))
                 (make-tree (list-ref list halfway)
                            (rec (list-head list halfway))
                            (rec (list-tail list (1+ halfway))))))))
  (rec list))
```

```
<<echo>>
<<tree-to-list>>
<<tree-makers>>

(let* ((nil '())
       (1-to-4 (list 1 2 3 4))
       (tree-right
        (rightward-tree 1-to-4))
       (tree-left
        (leftward-tree 1-to-4))
       (tree-middle
        (nested-tree (iota 10))))
  (echo "tree-right:" tree-right)
  (echo "tree-right list 1:" (tree->list-1 tree-right))
  (echo "tree-right list 2:" (tree->list-2 tree-right))
  (newline)
  (echo "tree-left:" tree-left)
  (echo "tree-left list 1:" (tree->list-1 tree-left))
  (echo "tree-left list 2:" (tree->list-2 tree-left))
  (newline)
  (echo "tree-middle:" tree-middle)
  (echo "tree-middle list 1:" (tree->list-1 tree-middle))
  (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

`ID: 6a43ff28-bb10-4e54-b692-59f258015af4`

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

`ID: dd34d29c-8a86-4455-84ea-5e56510c2dd4`

```
;; tree->list-1 evaluating the following tree:
;;     7
;;    / \
;;   3   9
;;  / \   \
;; 1   5   11
(append (append (append nil
                        (cons 1
                              nil))
                (cons 3
                      (append nil
                              (cons 5
                                    nil))))
        (cons 7
              (append nil
                      (cons 9
                            (append nil
                                    (cons 11
                                          nil))))))
;; holy cow that is wasteful. So many appends of nothing.

;; tree->list-2 evaluation (iterative)
(copy-to-list [full tree] nil)
(copy-to-list [tree 3 1 5]
              (cons 7
                    (copy-to-list
```

```
                     [tree 9 11]
                     nil)))
(copy-to-list [tree 3 1 5]
              (cons 7
                    (copy-to-list
                     nil
                     (cons 9
                           (copy-to-list
                            [tree 11]
                            nil)))))
(copy-to-list [tree 3 1 5]
              (cons 7
                    (copy-to-list
                     nil
                     (cons 9
                           (copy-to-list
                            nil
                            (cons 11
                                  (copy-to-list nil nil)))))))
(copy-to-list [tree 3 1 5]
              '(7 9 11))
(copy-to-list [tree 1]
              (cons 3
                    (copy-to-list
                     nil
                     (cons 5
                           (copy-to-list
                            nil
                            '(7 9 11))))))
(copy-to-list
 nil
 (cons 1
       (copy-to-list
        nil
        '(3 5 7 9 11))))
'(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

`ID: b748b886-879d-465d-87e6-c816cb02c4dd`

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.

```
<<make-tree>>
(define (list->tree elements)
  (car (partial-tree elements (length elements))))
(define (partial-tree elts n)
  (if (= n 0)
      (cons '() elts)
      (let ((left-size (quotient (- n 1) 2)))
        (let ((left-result
               (partial-tree elts left-size)))
          (let ((left-tree (car left-result))
                (non-left-elts (cdr left-result))
                (right-size (- n (+ left-size 1))))
            (let ((this-entry (car non-left-elts))
                  (right-result
                   (partial-tree
                    (cdr non-left-elts)
                    right-size)))
              (let ((right-tree (car right-result))
                    (remaining-elts
                     (cdr right-result)))
                (cons (make-tree this-entry
                                 left-tree
                                 right-tree)
                      remaining-elts))))))))
```

3.73.1 Question A

ID: bac079c0-fcdc-4ce9-ba35-22f22f0aeb7c

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

ID: 47ba4b88-3df4-40f4-ad3c-b7481e660ce1

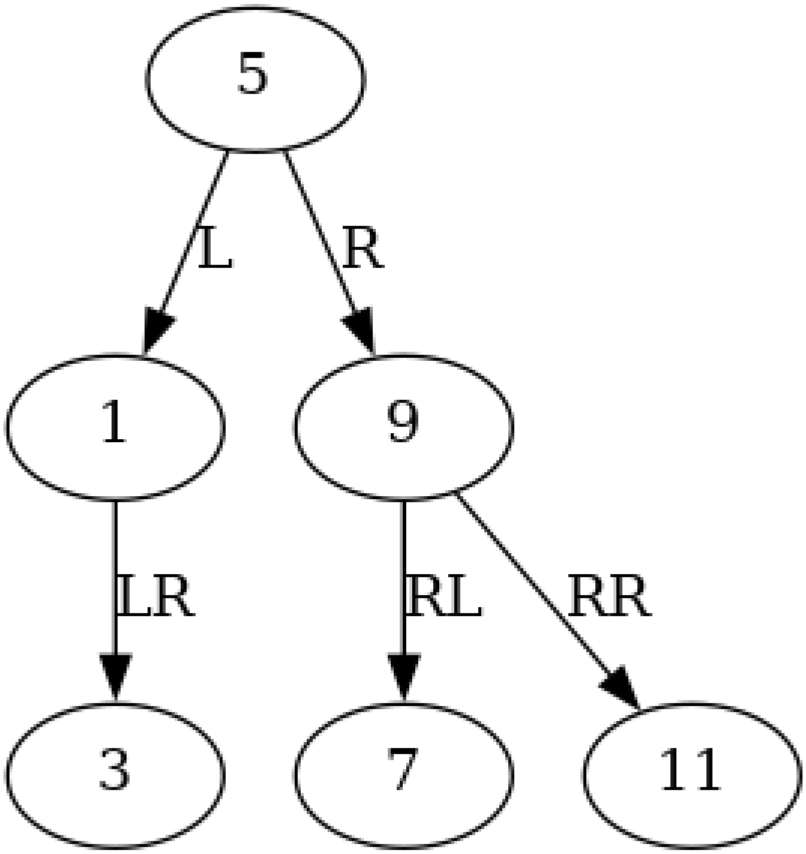
For my own sake, I'll reorganize this with `let*`.

```
(define (partial-tree elts n)
  (if (= n 0)
      (cons '() elts)
      (let* ((left-size (quotient (- n 1) 2))
             (left-result (partial-tree
                           elts
                           left-size))
```

```
              (left-tree (car left-result))
              (non-left-elts (cdr left-result))
              (right-size (- n (+ left-size 1)))
              (this-entry (car non-left-elts))
              (right-result (partial-tree
                             (cdr non-left-elts)
                             right-size))
              (right-tree (car right-result))
              (remaining-elts (cdr right-result)))
          (cons (make-tree this-entry
                           left-tree
                           right-tree)
                remaining-elts))))

```

Say we're evaluating `'(1 3 5 7 9 11)`. We define `left-size` as $\lfloor n/2 \rfloor$. `partial-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

`ID: c1c44972-e7a2-47ef-bd00-8a4ab612e98f`

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

`ID: 2d8751a2-ce68-4f57-adb3-3288009981c9`

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

`ID: adb5b85a-5b2d-4981-ae2a-ad03df26bbb1`

3.74.1 Textbook Definitions

ID: 0290682f-9b5f-41e8-8c2a-ca12aefab68d

```scheme
(define (element-of-set?-tree x set)
  (cond ((null? set) #f)
        ((= x (entry set)) #t)
        ((< x (entry set))
         (element-of-set?-tree
          x
          (left-branch set)))
        ((> x (entry set))
         (element-of-set?-tree
          x
          (right-branch set)))))
(define (adjoin-set-tree x set)
  (cond ((null? set) (make-tree x '() '()))
        ((= x (entry set)) set)
        ((< x (entry set))
         (make-tree
          (entry set)
          (adjoin-set-tree x (left-branch set))
          (right-branch set)))
        ((> x (entry set))
         (make-tree
          (entry set)
          (left-branch set)
          (adjoin-set-tree x (right-branch set))))))
```

3.74.2 Question

ID: 566edfbe-2c03-4fa3-a6bc-df4359dfb663

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

ID: 77574b95-ad69-4f55-8216-bc7bb3f30473

Ok, now that's stretching my brain for sure. Hmm...

I think I need to look closer at how `tree->list` works.

```scheme
(define (tree->list-debug tree)
  (define (copy-to-list tree result-list)
    (if (null? tree)
        result-list
        (let ((result
```

```
                (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 '()))
```

```
>> 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.

```
<<make-tree>>
<<list-to-tree>>
<<tree-to-list>>
<<tree-sets-txt>>

(define (tree->list-debug tree)
  (define (copy-to-list tree result-list)
    (if (null? tree)
        result-list
        (let ((result
               (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)
           (swap-trees))
          ((= (entry tree)
              (car result-list))
           (advance-tree-no-cons))
          ((= (entry tree) (entry held-tree))
           (advance-both-cons))
          ((< (entry tree) (entry held-tree))
           (advance-tree-cons))
          ((> (entry tree) (entry held-tree))
           (swap-trees))))
  (copy-to-list set1 set2 '()))

(define (union-set-tree-wrong2 set1 set2)
  (define (copy-to-tree tree result-tree)
    (if (null? tree)
        result-tree
        (copy-to-tree (left-branch tree)
                      (adjoin-set-tree (entry tree)
                                       (copy-to-tree
                                        (right-branch tree)
                                        result-tree)))))
  (copy-to-tree set1 set2))

(define (intersection-set-tree-wrong1 set1 set2)
  (define (copy-to-tree tree result-tree)
    (if (null? tree)
        result-tree
        (let ((adjoin-or-not
               (lambda (rest)
                 (if (element-of-set?-tree (entry tree)
                                           set2)
                     (adjoin-set-tree (entry tree)
                                      rest)
                     rest))))
          (copy-to-tree (left-branch tree)
                        (adjoin-or-not
                         (copy-to-tree
                          (right-branch tree)
```

```
                            result-tree))))))
      (copy-to-tree set1 '()))
```

```
<<sets-as-trees>>
<<echo>>
(load "mattcheck2.scm")
(let ((set1 (list->tree '(1 3 5 7 9 11)))
;; (5 (1 () (3 () ())) (9 (7 () ()) (11 () ())))
      (set2 (list->tree '(2 4 6 7 8 11)))
;; (6 (2 () (4 () ())) (8 (7 () ()) (11 () ())))
      (union (list->tree '(1 2 3 4 5 6 7 8 9 11)))
;; (5 (2 (1 () ()) (3 () (4 () ()))) (8 (6 () (7 () ())) (9 () (11 () ()))))
      (union-unbal '(6 (2 (1 () ()) (4 (3 () ()) (5 () ()))) (8 (7 () ()) (11 (9 () ()) ()))))
      (int-unbal '(11 (7 () ()) ()))
      (int (list->tree '(7 11))))
;; (7 () (11 () ()))
  (mattcheck "element-of-set?-tree true"
             (element-of-set?-tree 1 set1)
             #t)
  (mattcheck "element-of-set?-tree true"
             (element-of-set?-tree 7 set1)
             #t)
  (mattcheck "element-of-set?-tree false"
             (element-of-set?-tree 2 set1)
             #f)
  (mattcheck "union-set-tree-wrong2"
             (union-set-tree-wrong2 set1 set2)
             union-unbal)
  (mattcheck "intersection-set-tree-wrong1"
             (intersection-set-tree-wrong1 set1 set2)
             int-unbal)
  (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!

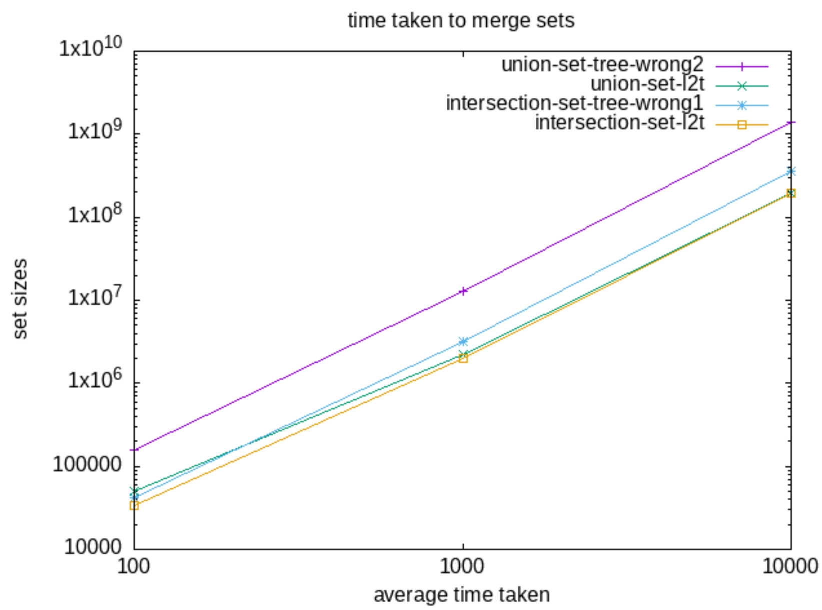
And... To my surprise, the answer most internet people have given is what I called the "wrong" one. In their case, using `tree->list` and `list->tree` to apply the list-based `union-set` and `intersection-set`, saying it's $\Theta(n)$. I also notice that they didn't use `element-of-set?` and `adjoin-to-set` like I did.

Let's compare.

```
(use-modules (ice-9 format))
(load "../mattbench.scm")
<<sets-txt>>
<<sets-as-trees>>
<<enumerate-interval>>
(define tree->list tree->list-2)
(define (union-set set1 set2)
  (cond ((or (null? set1) (null? set2))
         set2)
        ((element-of-set? (car set1) set2)
         (union-set (cdr set1)
                    set2))
        (else (cons (car set1)
                    (union-set (cdr set1)
                               set2)))))

;; from http://community.schemewiki.org/?sicp-ex-2.65
(define (union-set-l2t set1 set2)
  (cond ((null? set1) set2)
        ((null? set2) set1)
        (else (list->tree (union-set (tree->list set1) (tree->list set2))))))

(define (intersection-set-l2t set1 set2)
  (cond ((null? set1) '())
        ((null? set2) '())
        (else (list->tree (intersection-set (tree->list set1) (tree->list set2))))))

(define (make-2sets max)
  (let* ((a-start 0)
         (a-end 0.6)
         (b-start 0.4)
         (b-end 1))
    (cons (list->tree
           (enumerate-interval
            0
            (inexact->exact (* a-end max))))
          (list->tree
           (enumerate-interval
            (inexact->exact (* b-start max))
            (inexact->exact (* b-end max)))))))

(define (test-with 2sets repeats)
  (format #t "~&union-set-tree-wrong2: ~a"
          (cadr (mattbench2
                 (lambda()(union-set-tree-wrong2
                      (car 2sets)
```

```
                      (cdr 2sets)))
                 repeats)))
  (format #t "~&union-set-l2t: ~a"
          (cadr (mattbench2
                 (lambda()(union-set-l2t (car 2sets)
                                    (cdr 2sets)))
                 repeats)))
  (format #t "~&intersection-set-tree-wrong1: ~a"
          (cadr (mattbench2
                 (lambda()(intersection-set-tree-wrong1
                      (car 2sets)
                      (cdr 2sets)))
                 repeats)))
  (format #t "~&intersection-set-l2t: ~a"
          (cadr (mattbench2
                 (lambda()(intersection-set-l2t (car 2sets)
                                           (cdr 2sets)))
                 repeats))))
(format #t "~&~a" (make-2sets 10))
(define base-repeats 10000)
(test-with (make-2sets 100)
           base-repeats)
(test-with (make-2sets 1000)
           (/ base-repeats 2))
(test-with (make-2sets 10000)
           (/ 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
```



Not really enough datapoints, but you can at least see that the `l2t` variants are faster.

3.75 Exercise 2.66: binary tree `lookup`

ID: 7789edb1-b8d0-4587-b61b-4335abfc9485

3.75.1 Question

ID: 28c9b15a-b445-4efd-8a49-bb40f8cba291

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

ID: 5ef7e3f8-f4db-474d-8e9e-5fd60a22a411

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
            (key (entry tree-of-records)))
         (entry tree-of-records))
        ((< given-key
            (key (entry tree-of-records)))
         (lookup
          given-key
          (left-branch tree-of-records)))
        ((> given-key
            (key (entry tree-of-records)))
         (lookup
          given-key
          (right-branch tree-of-records)))))
```

```
<<lookup-tree>>
<<list-to-tree>>
<<echo>>
(load "mattcheck2.scm")
(let ((set1 (list->tree
             (list (make-record 1 'a)
                   (make-record 3 'b)
                   (make-record 5 'c)
                   (make-record 7 'd)
                   (make-record 9 'e)
                   (make-record 11 'f)))))
  (mattcheck "lookup retrieves correct record"
             (lookup 9 set1)
             (make-record 9 'e)))
```

```
SUCCEED at lookup retrieves correct record
```

3.76 Exercise 2.67: decoding Huffman tree messages

ID: b996c47a-9e4d-4f41-ae7a-ebfd48fa2be4

3.76.1 Text definitions

ID: 798df0ce-bc84-4113-9a95-5d6861590639

```
;; huffman-trees-txt
(define (make-leaf symbol weight)
  (list 'leaf
        symbol
```

```scheme
        weight))
(define (leaf? object)
  (eq? (car object)
       'leaf))
(define (symbol-leaf x)
  (cadr x))
(define (weight-leaf x)
  (caddr x))

(define (make-code-tree left right)
  (list left
        right
        (append (symbols left) (symbols right))
        (+ (weight left) (weight right))))

(define (left-branch  tree)
  (car tree))
(define (right-branch tree)
  (cadr tree))
(define (symbols tree)
  (if (leaf? tree)
      (list (symbol-leaf tree))
      (caddr tree)))
(define (weight tree)
  (if (leaf? tree)
      (weight-leaf tree)
      (cadddr tree)))
(define (decode bits tree)
  (define (decode-1 bits current-branch)
    (if (null? bits)
        '()
        (let ((next-branch
               (choose-branch (car bits) current-branch)))
          (if (leaf? next-branch)
              (cons (symbol-leaf next-branch)
                    (decode-1 (cdr bits) tree))
              (decode-1 (cdr bits) next-branch)))))
  (decode-1 bits tree))
(define (choose-branch bit branch)
  (cond ((= bit 0) (left-branch branch))
        ((= bit 1) (right-branch branch))
        (else (error ”bad bit: CHOOSE-BRANCH” bit))))
(define (adjoin-set x set)
  (cond ((null? set) (list x))
        ((< (weight x) (weight (car set))) (cons x set))
        (else (cons (car set)
                    (adjoin-set x (cdr set))))))
(define (make-leaf-set pairs)
```

```
  (if (null? pairs)
      '()
      (let ((pair (car pairs)))
        (adjoin-set (make-leaf (car pair)
                               (cadr pair))
                    (make-leaf-set (cdr pairs))))))
```

3.76.2 Question

ID: de18de6a-5926-4df4-aac7-ef6f625be186

Define an encoding tree and a sample message. Use the `decode` procedure to decode the message, and give the result.

3.76.3 Answer

ID: f3e5d321-b4a5-4e10-b4d6-f12b7905ff23

```
<<huffman-trees-txt>>
<<echo>>
(define sample-tree
  (make-code-tree (make-leaf 'A 4)
                  (make-code-tree
                   (make-leaf 'B 2)
                   (make-code-tree
                    (make-leaf 'D 1)
                    (make-leaf 'C 1)))))
(define whos-on-first-tree
  (make-code-tree
   (make-leaf 'who 8)
                  (make-code-tree
                   (make-leaf 'what 4)
                   (make-code-tree
                    (make-leaf 'Idontknow 2)
                    (make-leaf 'why 2)))))
(define sample-message '(0 1 1 0 0 1 0 1 0 1 1 1 0))
(define whos-on-first-message '(0 1 0 1 1 0 1 1 1))
(echo (decode sample-message sample-tree))
(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

ID: dad2b9f6-25ac-4de0-8d65-5ae120a639de

3.77.1 Question

ID: e10cf171-3127-4b9f-b240-0ad0b2db52ee

The encode procedure takes as arguments a message and a tree and produces the list of bits that gives the encoded message.

```
;; encode-txt
(define (encode message tree)
  (if (null? message)
      '()
      (append (encode-symbol (car message) tree)
              (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

ID: d20f06fa-24db-44eb-8aa8-7fc4382abbe3

```
;; encode-symbol
(define element-of-set? member)
(define (encode-symbol symbol tree)
  (define (rec t)
    (cond ((and (leaf? t)
                (eq? (symbol-leaf t)
                     symbol))
           '())
          ((element-of-set? symbol
                            (symbols (left-branch t)))
           (cons 0
                 (rec (left-branch t))))
          ((element-of-set? symbol
                            (symbols (right-branch t)))
           (cons 1
                 (rec (right-branch t))))
          (else (error "encode-symbol: logic error"))))
  (if (element-of-set? symbol (symbols tree))
      (rec tree)
      (error "encode-symbol: symbol not in tree")))
```

```
<<huffman-trees-txt>>
<<encode-txt>>
<<encode-symbol>>
(load "mattcheck2.scm")
(define sample-tree
  (make-code-tree (make-leaf 'A 4)
                  (make-code-tree
                   (make-leaf 'B 2)
                   (make-code-tree
                    (make-leaf 'D 1)
                    (make-leaf 'C 1)))))
(define whos-on-first-tree
  (make-code-tree
   (make-leaf 'who 8)
                  (make-code-tree
                   (make-leaf 'what 4)
                   (make-code-tree
                    (make-leaf 'Idontknow 2)
                    (make-leaf 'why 2)))))
(define sample-message '(0 1 1 0 0 1 0 1 0 1 1 1 0))
(define whos-on-first-message '(0 1 0 1 1 0 1 1 1))
(mattcheck "sample encoded message"
           sample-message
           (encode '(A D A B B C A)
                   sample-tree))
(mattcheck "who's on first encoded message"
           whos-on-first-message
           (encode '(who what Idontknow why)
                   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

ID: 08c0f6d4-2c04-4004-a7a4-6b22abba623a

3.78.1 Question

ID: 04f74d6f-2e1e-4bb2-925c-43eb2bca5a03

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.

```
(define (generate-huffman-tree pairs)
  (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

ID: fcbe5685-f1b4-4e54-a262-7c5378db7985

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.

```
;; generate-huffman-tree
(use-srfis '(1))
(define element-of-set? member)
(define (generate-huffman-tree pairs)
  (successive-merge (make-leaf-set pairs)))

(define (successivemerge-iter delayed leaves)
  (let ((dl (length delayed))
        (ll (length leaves)))
    (cond ((= ll 0)
           (successivemerge-iter '()
                                 delayed))
          ((= dl 0)
           (if (= ll 1)
               (car leaves)
               (successivemerge-iter (cons (car leaves) '())
                                     (cdr leaves))))
          ((and (= ll 1)
                (= dl 1))
           (make-code-tree (car delayed)
                           (car leaves)))
          ((<= (weight (car delayed))
               (weight (car leaves)))
           (successivemerge-iter (cdr delayed)
                                 (cons (make-code-tree (car delayed)
                                                       (car leaves))
                                       (cdr leaves))))
          ((> (weight (car delayed))
              (weight (car leaves)))
```

```
            (successivemerge-iter (cons (car leaves)
                                        delayed)
                                  (cdr leaves))))))

  (define (successive-merge leaves)
    (successivemerge-iter (cons (car leaves) '())
                          (cdr leaves)))
```

```
<<huffman-trees-txt>>
<<encode-txt>>
<<encode-symbol>>
<<generate-huffman-tree>>
<<echo>>
(load "mattcheck2.scm")

(let ((pairs
       (make-leaf-set '((A 8)(B 3)(C 1)(D 1)
                        (E 1)(F 1)(G 1)(H 1))))
      (answer
       (make-code-tree (make-leaf 'A 8)
                       (make-code-tree
                        (make-code-tree
                         (make-leaf 'B 3)
                         (make-code-tree
                          (make-leaf 'C 1)
                          (make-leaf 'D 1)))
                        (make-code-tree
                         (make-code-tree
                          (make-leaf 'E 1)
                          (make-leaf 'F 1))
                         (make-code-tree
                          (make-leaf 'G 1)
                          (make-leaf 'H 1)))))))
  (mattcheck "successive-merge"
             (successive-merge pairs)
             answer)
  (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.

```scheme
;; see 2.67
(define (adjoin-set x set)
  (cond ((null? set) (list x))
        ((< (weight x) (weight (car set)))
         (cons x set))
        (else (cons (car set)
                    (adjoin-set x (cdr set))))))
;; for my own edification
(define (adjoin-set-iter x set)
  (define (iter less more)
    (cond ((null? more)
           (reverse (cons x less)))
          ((< (weight x) (weight (car more)))
           (append (reverse (cons x less))
                   more))
          (else (adjoin-set-iter (cons (car more) less)
                                 (cdr more)))))

  (iter '() set))

;; https://codereview.stackexchange.com/a/117980
(define (successive-merge-small leaves)
  (if (null? (cdr leaves))
      (car leaves)
      (successive-merge-small
       (adjoin-set
        (make-code-tree (car leaves)
                        (cadr leaves))
        (cddr leaves)))))
```

Comparing my above code with the code for `adjoin-set`, mine is definitely unnecessarily complicated. How does the performance stack up?

```
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

ID: 315bef96-b4d4-41b1-ad8a-7d7b90a8fa96

3.79.1 Question

ID: 2a294aed-baaa-4e9c-9dbc-f66ddf33c7bf

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

ID: 180ceb80-51bc-4898-8b9c-bd6f712391e5

```scheme
(define hippie-set
  '((NA 16)(YIP 9)(SHA 3)(A 2)
    (GET 2)(JOB 2)(BOOM 1)(WAH 1)))
(define hippie-tree
  (generate-huffman-tree hippie-set))
(define hippie-plaintext
  '(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))
(define hippie-encoded
  (encode hippie-plaintext
          hippie-tree))
```

```scheme
<<huffman-trees-txt>>
<<encode-txt>>
<<encode-symbol>>
<<generate-huffman-tree>>
(define (successive-merge leaves)
  (if (null? (cdr leaves))
      (car leaves)
      (successive-merge
       (adjoin-set
        (make-code-tree (car leaves)
                        (cadr leaves))
        (cddr leaves)))))
<<hippie-trees>>
<<echo>>
(load "mattcheck2.scm")

(echo "Length of unencoded message is" (length hippie-plaintext)
↪  "words.")
(echo "Length of encoded message is" (length hippie-encoded) "bits")
(echo "Unencoded message (newlines added):" (decode hippie-encoded
↪  hippie-tree))
```

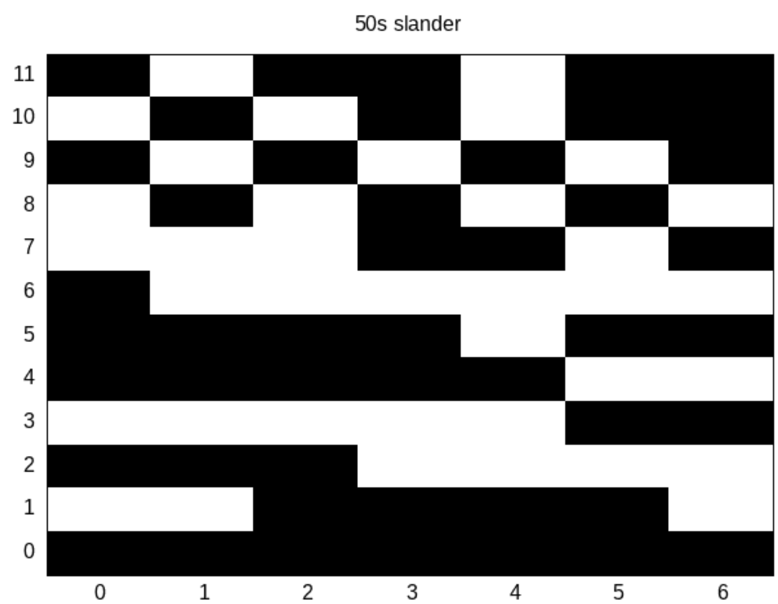
```
Length of unencoded message is 36 words.
Length of encoded message is 84 bits
Unencoded message (newlines added):
(GET A JOB
SHA NA NA NA NA NA NA NA NA
GET A JOB SHA NA NA NA NA NA NA NA NA
WAH YIP YIP YIP YIP
YIP YIP YIP YIP YIP
SHA BOOM)
```

A fixed-length alphabet would take 3 bits per word, or 108 bits. So the VLE is 77% the size of anything else.

```scheme
<<huffman-trees-txt>>
<<encode-txt>>
<<encode-symbol>>
<<generate-huffman-tree>>
(define (successive-merge leaves)
  (if (null? (cdr leaves))
      (car leaves)
      (successive-merge
       (adjoin-set
        (make-code-tree (car leaves)
                        (cadr leaves))
        (cddr leaves)))))
<<hippie-trees>>
```

```
(define (split-list-every-x list x)
  (define (rec ll)
    (if (< (length ll) x)
        (cons ll '())
        (cons (list-head ll x)
              (rec (list-tail ll x)))))
  (rec list))
(split-list-every-x hippie-encoded 7)
```

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



3.80 Exercise 2.71

ID: b5005366-1d37-4ed6-89a5-bc8634e54d57

3.80.1 Questions

ID: 8eb2252c-8049-4cea-bdb5-792718bc1a77

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

ID: 8e12f06c-63c2-4ec4-9616-698443a28b7f

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`

ID: 39a59b4b-6baf-4145-ab76-ba936dc46191

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)
                            (symbols (left-branch t)))
           (cons 0 ;; O(1)
                 (rec (left-branch t)))) ;; rec is O(log symbols)
          ((element-of-set? symbol ; O(symbols of right-branch)
                            (symbols (right-branch t)))
           (cons 1 ;; O(1)
                 (rec (right-branch t)))) ;; rec is O(log symbols)
          (else (error "encode-symbol: logic error"))))
  (if (element-of-set? symbol (symbols tree)) ;; O(symbols)
      (rec tree) ;; rec is O(log n)
      (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

`ID: c3c0cc87-1e03-41c1-995d-4656c2d60e9d`

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

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)))
```

[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.
[8] https://www.gnu.org/software/guile/manual/html_node/Hash-Table-Reference.html

```
;; 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

ID: 4e21bd2c-be0f-45ed-9721-d185ca6173aa

2.3.2: Example: Symbolic differentiator described a program that performs symbolic differentiation:

```
(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))))
        ;; more rules can be added here
        (else (error ”unknown expression type:
                      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

```
(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))))
(define (operator exp) (car exp))
(define (operands exp) (cdr exp))
```

3.83.1 Question A

`ID: 1e6e078f-5590-4b44-b5c0-29ac728a95a3`

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

`ID: bfd90785-9393-464a-87ac-a742a38b183c`

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

`ID: 9d2c846d-4a29-4663-a8f0-4d65bd18d3c9`

- 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

ID: c1d2539e-32a5-4253-89dc-a007380a8d0a

Here's my first solution, which is functional, but not at all idiomatic like the textbook's examples of such a system.

```
(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
(put 'deriv '+
     (lambda (ops var)
       (make-sum (deriv (addend ops) var)
```

```
                 (deriv (augend ops) var))))
(put 'deriv '*
     (lambda (ops var)
       (make-sum (make-product
                  (multiplier ops)
                  (deriv (multiplicand ops) var))
                 (make-product
                  (deriv (multiplier ops) var)
                  (multiplicand ops)))))
(put 'deriv '**
     (lambda (ops var)
         (make-product
          (make-product
           (exponent ops)
           (make-exponentiation (base ops)
                                (make-sum (exponent ops) -1)))
          (deriv (base ops) var))))
```

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

(mattcheck "deriv: exponentiation 1"
           (deriv (make-exponentiation 'a 5) 'a)
           '(* 5 (** a 4)))
(mattcheck "deriv: exponentiation 2"
           (deriv (make-exponentiation 'a 'b) 'a)
           '(* 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

ID: 816f36f8-f2e3-40e0-b8af-3a0374b47b02

```
(define (deriv exp var)
  (define (variable? x) (symbol? x))
  (define (same-variable? v1 v2)
```

```
    (and (variable? v1) (variable? v2) (eq? v1 v2)))
  (define (operator exp) (type-tag exp)) ;; use type tags
  (define (operands exp) (contents exp))

  (cond ((number? exp) 0)
        ((variable? exp) (if (same-variable? exp var) 1 0))
        (else ((get 'deriv (operator exp))
               (operands exp) var))))

(define (install-deriv-package choice)
  ;;shared procedures
  (define (=number? exp num)
    (and (number? exp) (= exp num)))
  (define (make-sum a1 a2)
    (cond ((=number? a1 0) a2)
          ((=number? a2 0) a1)
          ((and (number? a1) (number? a2))
           (+ a1 a2))
          (else (attach-tag '+ (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 (attach-tag '* (list m1 m2)))))
  (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 (attach-tag '** (list b e)))))

  ;; packages
  (define (install-sum-package)
    (define (addend s) (car s))
    (define (augend s) (cadr s))
    (put 'deriv '+
         (lambda (ops var)
           (make-sum (deriv (addend ops) var)
                     (deriv (augend ops) var)))))

  (define (install-mult-package)
    (define (multiplier p) (car p))
    (define (multiplicand p) (cadr p))
    (put 'deriv '*
         (lambda (ops var)
           (make-sum (make-product
                      (multiplier ops)
```

```scheme
                        (deriv (multiplicand ops) var))
                       (make-product
                        (deriv (multiplier ops) var)
                        (multiplicand ops))))))
  (define (install-exp-package)
    (define (base e) (car e))
    (define (exponent e) (cadr e))
    (put 'deriv '**
         (lambda (ops var)
           (make-product
            (make-product
             (exponent ops)
             (make-exponentiation (base ops)
                                  (make-sum (exponent ops)
                                            -1)))
            (deriv (base ops) var)))))

  ;;body
  (cond ((eqv? choice 'sum)
         (install-sum-package))
        ((eqv? choice 'mult)
         (install-mult-package))
        ((eqv? choice 'exp)
         (install-exp-package))))
```

```scheme
<<op-table>>
<<type-tagging>>
<<deriv-datadriven2>>
(load "mattcheck2.scm")
(install-deriv-package 'sum)
(install-deriv-package 'mult)
(install-deriv-package 'exp)
(mattcheck "deriv: basic addition"
           (deriv '(+ x 3) 'x)
           1)
(mattcheck "deriv: basic multiplication"
           (deriv '(* x y) 'x)
           'y)

(mattcheck "deriv: exponentiation 1"
           (deriv '(** a 5) 'a) ;; no more external access
           '(* 5 (** a 4)))
(mattcheck "deriv: exponentiation 2"
           (deriv '(** a b) 'a)
           '(* 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

`ID: 21a3dd2e-03a3-4f11-9602-fa2138d03614`

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

```scheme
((get (operator exp) 'deriv)
 (operands exp) var)
```

What corresponding changes to the derivative system are required?

3.83.7 Answer D

`ID: 5e34ebcd-6c84-426b-a4fc-55936d499a1a`

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

`ID: 394db043-6005-497c-aefe-65664c787b0f`

3.84.1 Questions ABC

`ID: 0bf46397-0039-4444-9cd5-66e4d57b2993`

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

ID: bde07dab-86b0-4f2d-97f8-6ae7f6d69f11

Let's talk assumptions:

- Assume every division has one file
- Assume HQ wants a consistent schema returned

```
<<make-tree>>
<<make-record>>
<<op-table>>
(define (find x ll)
  (define (iter ll) ;; aka SRFI-1 'find'
    (cond ((null? ll)
           #f)
          ((equal? id (employee-id-east (car ll)))
           (car ll))
          (else (iter (cdr ll)))))
  (iter ll))

;; East: records as named cdr aliases
(define (install-east-package)
  (define (make-employee-east id salary)
```

```
    (cons id (/ salary 1000))) ;; East keeps salary in thousands
  (define (employee-id-east record)
    (car record))
  (define (employee-salary-east record)
    (* (cdr record) 1000))
  (define East-Employee-List
    (list (make-employee-east 101 37000) ; bob
          (make-employee-east 102 48000) ; linda
          (make-employee-east 103 29000) ; john
          ))
  (define (find-employee-east id)
    (find id East-Employee-List))
  (put 'employee-record 'East
       (lambda (id)
         (let ((e (find-employee-east id)))
           (if e
               (list (list 'id id)
                     (list (employee-salary-east e)))
               #f))))
  (put 'employee-salary 'East
       (lambda(id)
         (employee-salary-east (find-employee-east id)))))

;; records as lookup trees (Ex 2.66)
(define (install-west-package)
  (define (maketree . args)
    (apply list->tree args))
  (define (make-employee-west id name salary)
    (make-record 'id
                 (maketree (make-record 'name name)
                           (make-record 'salary salary))))
  (define West-Record-File
    (maketree (make-employee-west 201 "James" 55000)
              (make-employee-west 202 "Joy" 45000)
              (make-employee-west 203 "Richard" 73000)))
  (define (employee-west id)
    (lookup id West-Record-File))
  (define (employee-attrib-west attrib record)
    (lookup attrib record))
  (put 'employee-record 'West
       (lambda(id)
         (let ((e (employee-west id)))
           (if e
               (list (list 'id id)
                     (list 'name (employee-attrib-west 'name e))
                     (list 'salary (employee-attrib-west 'salary e)))
               #f))))
  (put 'employee-salary 'West
```

```
       (lambda(id)
         (employee-attrib-west 'salary
                               (employee-west id)))))

;; South: records as lambdas
(define (install-south-package)
  (define (make-employee-south id name salary)
    (lambda(value)
      (cond ((eqv? 'id) id)
            ((eqv? 'name) name)
            ((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
```

```scheme
;; 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

`ID: 2aae8544-ff80-43e7-8326-e15c9ae2b4e3`

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

`ID: c436ede2-c1b0-4d63-bb88-09dc8e628301`

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

`ID: 8d2953ca-9c50-48b9-9fea-f1c4debc5109`

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

`ID: b20622b4-2811-4ee0-9762-76072a23893c`

3.86.1 Question

ID: 642c7cfe-cff7-4534-8350-dd8ecff11555

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

ID: d54e5762-74e3-41b4-b389-2c3996b91dd6

```
(define (make-from-mag-ang r a)
  (define (dispatch op)
    (cond ((eq? op 'magnitude) r)
          ((eq? op 'angle) a)
          ((eq? op 'real-part)
           (* r (cos a)))
          ((eq? op 'imag-part)
           (* r (sin a)))
          (else (error "Unknown op: MAKE-FROM-MAG-ANG" op))))

  dispatch)
```

3.87 Exercise 2.76: comparing generic operation solutions

ID: f56af310-38ac-4949-b7cb-b0a0d395ac43

3.87.1 Question

ID: c4940adc-aea1-4c5a-b331-3a25eab26018

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

ID: 19f0c03b-522f-423a-a39b-a19e1a91560a

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

`ID: 1aba7a5c-481a-4903-8ae0-46e0f6c55dd0`

3.88.1 Question

`ID: e1caa9bd-b3a1-4713-ad39-68c8e292c703`

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 an error message from `apply-generic`, saying there is no method for the operation `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":

```
(put 'real-part '(complex) real-part)
(put 'imag-part '(complex) imag-part)
(put 'magnitude '(complex) magnitude)
(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?

[9] http://c2.com/cgi/wiki?ExpressionProblem

[10] http://community.schemewiki.org/?sicp-ex-2.76

3.88.2 Answer

ID: 48f39e76-e54e-4d18-b482-704d7aaa56db

Surprised to find I never actually imported `apply-generic`.

```
<<op-table>>
(define (apply-generic op . args)
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
          (apply proc (map contents args))
          (error
            "No method for these types: APPLY-GENERIC"
            (list op type-tags))))))
```

```
(define (square x) (* x 2))
(define (add x y) (apply-generic 'add x y))
(define (sub x y) (apply-generic 'sub x y))
(define (mul x y) (apply-generic 'mul x y))
(define (div x y) (apply-generic 'div x y))
(define (real-part z) (apply-generic 'real-part z))
(define (imag-part z) (apply-generic 'imag-part z))
(define (magnitude z) (apply-generic 'magnitude z))
(define (angle z) (apply-generic 'angle z))

(define (install-rational-package)
  ;; internal procedures
  (define (numer x) (car x))
  (define (denom x) (cdr x))
  (define (make-rat n d)
    (if (or (inexact? n)
            (inexact? d))
        (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))
  (define (make-from-real-imag x y) (cons x y))
  (define (magnitude z)
    (sqrt (+ (square (real-part z))
             (square (imag-part z)))))
  (define (angle z)
    (atan (imag-part z) (real-part z)))
  (define (make-from-mag-ang r a)
    (cons (* r (cos a)) (* r (sin a))))
  ;; interface to the rest of the system
  (define (tag x) (attach-tag 'rectangular x))
  (put 'real-part '(rectangular) real-part)
  (put 'imag-part '(rectangular) imag-part)
  (put 'magnitude '(rectangular) magnitude)
  (put 'angle '(rectangular) angle)
  (put 'make-from-real-imag 'rectangular
```

```scheme
       (lambda (x y) (tag (make-from-real-imag x y))))
  (put 'make-from-mag-ang 'rectangular
       (lambda (r a) (tag (make-from-mag-ang r a))))
  'done)

(define (install-polar-package)
  ;; internal procedures
  (define (magnitude z) (car z))
  (define (angle z) (cdr z))
  (define (make-from-mag-ang r a) (cons r a))
  (define (real-part z) (* (magnitude z) (cos (angle z))))
  (define (imag-part z) (* (magnitude z) (sin (angle z))))
  (define (make-from-real-imag x y)
    (cons (sqrt (+ (square x) (square y)))
          (atan y x)))
  ;; interface to the rest of the system
  (define (tag x) (attach-tag 'polar x))
  (put 'real-part '(polar) real-part)
  (put 'imag-part '(polar) imag-part)
  (put 'magnitude '(polar) magnitude)
  (put 'angle '(polar) angle)
  (put 'make-from-real-imag 'polar
       (lambda (x y) (tag (make-from-real-imag x y))))
  (put 'make-from-mag-ang 'polar
       (lambda (r a) (tag (make-from-mag-ang r a))))
  'done)

(define (install-complex-package)
  ;; imported procedures from rectangular and polar packages
  (define (make-from-real-imag x y)
    ((get 'make-from-real-imag 'rectangular) x y))
  (define (make-from-mag-ang r a)
    ((get 'make-from-mag-ang 'polar) r a))
  ;; 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))
```

```
(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)
       (lambda (x y) (tag (/ x y))))
  (put 'make 'scheme-number (lambda (x) (tag x)))
  'done)
```

First I want to do sanity checks on the system.

```
<<type-tagging>>
<<generic-arithmetic>>
<<scheme-number-package>>
<<echo>>
(load "mattcheck2.scm")

(install-rational-package)
(install-rectangular-package)
(install-polar-package)
```

```
(install-complex-package)
(install-scheme-number-package)

<<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:

```
,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 procedures 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

ID: b6c4fadc-2d78-4784-8975-91661ee5c2d1

3.89.1 Question

ID: 06442f66-6f3d-490b-9b82-a5f9a2f9fb9f

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

ID: 53716102-7937-4d59-9d2d-3be9a511caa4

```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))
  (echo "3m+4deg - 3m+4deg =" (sub p p))
  (echo "3m+4deg * 3m+4deg =" (mul p p))
  (echo "3m+4deg / 3m+4deg =" (div p p)))
```

```
<<sqrt>>
<<square>>
<<echo>>
<<smarter-type-tagging>>
<<generic-arithmetic>>
<<scheme-number-package>>
(load "mattcheck2.scm")

(install-rational-package)
(install-rectangular-package)
(install-polar-package)
(install-complex-package)
(install-scheme-number-package)

<<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

ID: 9f746c37-73b7-4a0b-b5a4-ef7a6c84c573

3.90.1 Question

ID: fca46553-9125-4444-9397-9ce4f275660a

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

ID: c7860a27-c8ec-4d8e-9ed7-3d4aeeac3b95

```scheme
(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 (= (apply-generic 'magnitude x)
                          (apply-generic 'magnitude y))
                       (= (apply-generic 'angle x)
                          (apply-generic 'angle y))))
                 (else
                  (and (= (apply-generic 'real-part x)
                          (apply-generic 'real-part y))
                       (= (apply-generic 'imag-part x)
                          (apply-generic 'imag-part y))))))))
  (put 'equ? '(rational rational)
       (lambda(x y)
         (and (= ((get 'numer '(rational)) x)
                 ((get 'numer '(rational)) y))
              (= ((get 'denom '(rational)) x)
                 ((get 'denom '(rational)) y)))))
  (put 'equ? '(real real)
       (lambda(x y) (= x y))))
```

```scheme
<<sqrt>>
<<square>>
<<echo>>
<<smarter-type-tagging>>
<<generic-arithmetic>>
<<scheme-number-package>>
<<equ?>>
(load "mattcheck2.scm")

(install-rational-package)
(install-rectangular-package)
(install-polar-package)
(install-complex-package)
(install-scheme-number-package)
(install-equ?)

```

```
<<gen-math-tests>>

(mattcheck "basic equ? true" (equ? 5 5) #t)
(mattcheck "basic equ? false" (equ? 4 5) #f)
(mattcheck "rect equ? true" (equ? (make-complex-from-real-imag 2 3)
                                  (make-complex-from-real-imag 2 3)) #t)
(mattcheck "rect equ? false" (equ? (make-complex-from-real-imag 2 3)
                                   (make-complex-from-real-imag 2 5)) #f)
(mattcheck "polar equ? true" (equ? (make-complex-from-mag-ang 2 3)
                                   (make-complex-from-mag-ang 2 3)) #t)
(mattcheck "polar equ? false" (equ? (make-complex-from-mag-ang 2 3)
                                    (make-complex-from-mag-ang 2 5)) #f)
(mattcheck "rational equ? true" (equ? (make-rational 2 3)
                                      (make-rational 2 3)) #t)
(mattcheck "rational equ? false" (equ? (make-rational 2 3)
                                       (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

ID: fe2432b6-f0a4-4691-9cf4-456d9542d1ee

3.91.1 Question

ID: c2cd7c47-7c6e-43d6-9094-adbb67edb78f

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

ID: 69ef5aa7-491e-41e2-9b2c-96b93dfa4594

```
(define (=zero? x) (apply-generic '=zero? x))
(define (install-=zero?)
  (put '=zero? '(scheme-number)
       (lambda(x) (= x 0)))
  (put '=zero? '(complex)
```

```
       (lambda(x)
         (apply-generic '=zero? x)))
  (let ((magnitude (get 'magnitude '(polar))))
    (put '=zero? '(polar)
         (lambda(x)
           (= 0 (magnitude x)))))
  (let ((real-part (get 'real-part '(rectangular)))
        (imag-part (get 'imag-part '(rectangular))))
    (put '=zero? '(rectangular)
         (lambda(x)
           (and (= 0 (real-part x))
                (= 0 (imag-part x))))))
  (put '=zero? '(rational)
       (lambda(x)
         (or (= 0 ((get 'numer '(rational)) x))
             (= 0 ((get 'denom '(rational)) x))))))
```

```
<<sqrt>>
<<square>>
<<echo>>
<<smarter-type-tagging>>
<<generic-arithmetic>>
<<scheme-number-package>>
<<equ?>>
<<eqzero?>>
(load "mattcheck2.scm")

(install-rational-package)
(install-rectangular-package)
(install-polar-package)
(install-complex-package)
(install-scheme-number-package)
(install-equ?)
(install-=zero?)
<<gen-math-tests>>

(mattcheck "basic =zero? true" (=zero? 0) #t)
(mattcheck "basic =zero? false" (=zero? 5) #f)
(mattcheck "rect =zero? true" (=zero? (make-complex-from-real-imag 0 0)) #t)
(mattcheck "rect =zero? false" (=zero? (make-complex-from-real-imag 2 5)) #f)
(mattcheck "polar =zero? true" (=zero? (make-complex-from-mag-ang 0 3)) #t)
(mattcheck "polar =zero? false" (=zero? (make-complex-from-mag-ang 2 0)) #f)
```

```scheme
(mattcheck "rational =zero? true" (=zero? (make-rational 0 3)) #t)
(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

ID: a05905d5-b680-4c93-b5fa-00fb5b961efd

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.

```scheme
(define (apply-generic op . args)
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
          (apply proc (map contents args))
          (if (= (length args) 2)
              (let ((type1 (car type-tags))
                    (type2 (cadr type-tags))
                    (a1 (car args))
                    (a2 (cadr args)))
                (let ((t1->t2
                       (get-coercion type1
                                     type2))
                      (t2->t1
                       (get-coercion type2
                                     type1)))
                  (cond (t1->t2
                         (apply-generic
                          op (t1->t2 a1) a2))
                        (t2->t1
                         (apply-generic
```

```
                          op a1 (t2->t1 a2)))
                        (else
                         (error
                          "No method for
                           these types"
                          (list
                           op
                           type-tags))))))
              (error
               "No method for these types"
               (list op type-tags)))))))
```

This assumes `put-coercion` and `get-coercion`.

```
(define type-table (make-hash-table))

(define (put-coercion type1 type2 f)
  (hash-set! type-table (cons type1 type2) f))
(define (get-coercion type1 type2)
  (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`

`ID: 4f44b71f-1d6d-4776-829e-4475404c5fb3`

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:

```
(define (scheme-number->scheme-number n) n)
(define (complex->complex z) z)
(put-coercion 'scheme-number
              'scheme-number
              scheme-number->scheme-number)
(put-coercion 'complex 'complex complex->complex)
```

3.93.1 Question A

`ID: 00de09fa-c220-4daf-a59e-6334cc49d2f6`

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:

```
(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:

```
;; following added to Scheme-number package
(put 'exp '(scheme-number scheme-number)
     (lambda (x y) (tag (expt x y))))
     ; using primitive expt
```

What happens if we call `exp` with two complex numbers as arguments?

3.93.2 Answer A

`ID: 7102b5d7-b1a0-4348-bb55-9c7e0a193359`

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

`ID: 72dd7fdb-49d3-46f6-86f4-81faecac354b`

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

`ID: 3afc606b-b683-45fd-84dc-beefcb87f989`

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

`ID: e037f8aa-5896-4027-a672-1f2716061497`

Modify `apply-generic` so that it doesn't try coercion if the two arguments have the same type.

3.93.6 Answer C

ID: bd571c23-5886-4424-bd91-96fd8b9125c1

```
(define (apply-generic op . args)
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
          (apply proc (map contents args))
          (if (= (length args) 2)
              (let ((type1 (car type-tags))
                    (type2 (cadr type-tags))
                    (a1 (car args))
                    (a2 (cadr args)))
                (if (equal? type1 type2)
                  ;      'failgood ; proving correct operation
                        (error
                     "No method for this type"
                     (list op type-tags))
                    (let ((t1->t2
                           (get-coercion type1
                                         type2))
                          (t2->t1
                           (get-coercion type2
                                         type1)))
                      (cond (t1->t2
                             (apply-generic
                              op (t1->t2 a1) a2))
                            (t2->t1
                             (apply-generic
                              op a1 (t2->t1 a2)))
                            (else
                             (error
                              "No coercion method for these types"
                              (list
                               op
                               type-tags)))))))
              (error
               "No method for these types"
               (list op type-tags)))))))
```

```
(put-coercion 'scheme-number 'rational
              (lambda(num)
                (make-rational num 1)))
<<eqnumber>>
(put-coercion 'rational 'complex
              (lambda(num)
                (make-complex-from-real-imag
```

```
                   (if (=number? 1 ((get 'denom '(rational)) num)) ; if rat could be integer
                       ((get 'numer '(rational)) num)       ; might as well simplify
                       num) 0)))
(put-coercion 'scheme-number 'complex
              (lambda(num)
                (make-complex-from-real-imag num 0)))
```

```
<<op-table>>
<<type-table>>
<<sqrt>>
<<square>>
<<echo>>
<<smarter-type-tagging>>
<<apply-generic-coerce-fixed>>
<<generic-arithmetic>>
<<scheme-number-package>>
<<equ?>>
<<eqzero?>>
(load "mattcheck2.scm")

(install-rational-package)
(install-rectangular-package)
(install-polar-package)
(install-complex-package)
(install-scheme-number-package)
(install-equ?)
(install-=zero?)
<<gen-math-tests>>
<<basic-coercions>>

(mattcheck "good coercion test"
           (mul (make-rational 8 1)
                (make-rational 2 3))
           (mul 8 (make-rational 2 3)))
(mattcheck "bad coercion test"
           (apply-generic 'foobar 8 9)
           'failgood)
```

```
SUCCEED at good coercion test
SUCCEED at bad coercion test
```

3.94 Exercise 2.82: `apply-generic` with multiple coerced arguments

ID: eb0495b5-c171-40a9-8934-569e450dc3d8

3.94.1 Question

ID: 01e6a7c6-23a3-439b-ac95-233a870b2f1d

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

ID: 964bfcf4-d8d5-4acb-942e-b34e11b4c04a

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:

```
(use-srfis '(1))
;; Version of apply-generic that look for coercion functions
;; that work for the entire list
(define (apply-generic op . args)
  (define types (map type-tag args))
  (define (coercions-for-list ll target)
    (map (lambda(item) ; find coercion functions, if identical type
    ↪  return id func
           (let ((type (type-tag item)))
             (if (equal? type target)
                 (lambda(x)x)
                 (get-coercion type target))))
         args))
  ;; map-coercions: list of types -> list of tagged numbers.
  ;; find one type which all items can be coerced to,
  ;; then do all coercions and return as list
  (define (map-coercions ll)
    (if (null? ll) ;; exhausted possibilities
        (error
         "No method between all these types"
         (list op types))
```

```
        (let ((tries (coercions-for-list ll (car ll))))
          (if (every procedure? tries); if all coercions found, apply
          ↪  them
              (map (lambda(f x) (f x))
                   tries args)
              (map-coercions (cdr ll)))))) ; else keep trying
  (let ((proc (get op types)))
    (if proc
        (apply proc (map contents args))
        (let ((coerced (map-coercions types)))
          (apply (get op (map type-tag coerced))
                 (map contents coerced))))))
```

Testing my code:

```
<<op-table>>
<<type-table>>
<<sqrt>>
<<square>>
<<echo>>
<<smarter-type-tagging>>
<<apply-generic-coerce-varargs-naive>>
<<generic-arithmetic>>
<<scheme-number-package>>
<<equ?>>
<<eqzero?>>
(load "mattcheck2.scm")

(install-rational-package)
(install-rectangular-package)
(install-polar-package)
(install-complex-package)
(install-scheme-number-package)
(install-equ?)
(install-=zero?)
<<gen-math-tests>>
<<basic-coercions>>

(mattcheck "num -> frac coercion test"
           (mul (make-rational 8 1)
                (make-rational 2 3))
           (mul 8 (make-rational 2 3)))
(mattcheck "num -> complex coercion test"
           (mul (make-complex-from-real-imag 8 0)
                (make-complex-from-real-imag 2 3))
           (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.

```
(use-srfis '(1))
;; Version of apply-generic that tries to coerce each variable
;; into each type represebted in args.
(define (apply-generic op . args)
  (define types (map type-tag args))
  ;; map-coercion: type, list of tagged numbers -> list of tagged numbers
  ;; try to coerce all items in list to given type. If this isn't possible,
  ;; leave the item alone. Meant for fold.
  (define (map-coercion type-to-try items)
    (map (lambda(item) ; find and apply coercion functions,
           (let* ((coerce (get-coercion (type-tag item) type-to-try)))
             (cond ((or (not coerce)
                        (equal? (type-tag item) type-to-try))
                    item)
                   (else (coerce item)))))
         items))

  (let ((proc (get op types)))
    (if proc ; is this already doable?
        (apply proc (map contents args))
        ; if not, try type coercion
        (let ((coerced (fold map-coercion args types)))
          (apply (get op (map type-tag coerced))
                 (map contents coerced))))))
;; However this won't actually handle multiple arguments
```

```
<<op-table>>
<<type-table>>
<<sqrt>>
<<square>>
<<echo>>
<<smarter-type-tagging>>
<<apply-generic-coerce-less-naive>>
<<generic-arithmetic>>
<<scheme-number-package>>
```

[11]starting on line 269 https://github.com/danielpi/SICP-Exercises/blob/46a6b38fd83faae21db239f40d913ff24062d2e6/Racket/Ex%202.82%20Generalized%20apply-generic.rkt#L269

```
<<equ?>>
<<eqzero?>>
(load "mattcheck2.scm")

(install-rational-package)
(install-rectangular-package)
(install-polar-package)
(install-complex-package)
(install-scheme-number-package)
(install-equ?)
(install-=zero?)
<<gen-math-tests>>
<<basic-coercions>>

(mattcheck "num -> frac coercion test"
           (mul (make-rational 8 1)
                (make-rational 2 3))
           (mul 8 (make-rational 2 3)))
(mattcheck "num -> complex coercion test"
           (mul (make-complex-from-real-imag 8 0)
                (make-complex-from-real-imag 2 3))
           (mul 8 (make-complex-from-real-imag 2 3)))
(mattcheck "multiple arguments test"
           (false-if-exception (apply-generic 'add 8 (make-rational 2 3)
                                              (make-complex-from-real-imag 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"
```

3.94.3 Answer with vararg support

ID: b1595a6e-8318-4314-819c-97f62ea09b03

```
(use-srfis '(1))
;; Version of apply-generic that tries to coerce each variable
;; into each type represebted in args.
(define (apply-generic op . args)
  (define types (map type-tag args))
  ;; map-coercion: type, list of tagged numbers -> list of tagged numbers
  ;; try to coerce all items in list to given type. If this isn't possible,
  ;; leave the item alone. Meant for fold.
  (define (map-coercion type-to-try items)
```

```
    (map (lambda(item) ; find and apply coercion functions,
           (let* ((coerce (get-coercion (type-tag item) type-to-try)))
             (cond ((or (not coerce)
                        (equal? (type-tag item) type-to-try))
                    item)
                   (else (coerce item)))))
         items))

  (let ((proc (get op types)))
    (if proc ; is this already doable?
        (apply proc (map contents args))
        ; if not, try type coercion
        (let ((coerced (fold map-coercion args types)))
          (if (< (length args) 3)
              ;; short args
              (apply (get op (map type-tag coerced))
                     (map contents coerced))
              ;; long args
              (reduce (lambda(elem previous)
                        (apply-generic op previous elem))
                      "ERROR" coerced))))))
```

```
<<op-table>>
<<type-table>>
<<sqrt>>
<<square>>
<<echo>>
<<smarter-type-tagging>>
<<apply-generic-coerce-varargs>>
<<generic-arithmetic>>
<<scheme-number-package>>
<<equ?>>
<<eqzero?>>
(load "mattcheck2.scm")

(install-rational-package)
(install-rectangular-package)
(install-polar-package)
(install-complex-package)
(install-scheme-number-package)
(install-equ?)
(install-=zero?)
<<basic-coercions>>

(mattcheck "multiple arguments test"
           (apply-generic 'add 8 2 3)
           13)
(mattcheck "multiple arguments test 2"
```

```
           (apply-generic 'add (make-complex-from-real-imag 3 5)
                          (make-complex-from-real-imag 3 5)
                          (make-complex-from-real-imag 3 5))
           (cons 'complex
                 (cons 'rectangular
                       (cons 9 15))))
 (mattcheck "multiple arguments coercion test"
           (false-if-exception
            (apply-generic 'add 8 (make-rational 2 3)
            (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`

ID: 26a42618-c607-4abc-a7cc-affbf2779db7

3.95.1 Question

ID: 3bd63c9b-5ff5-4e04-8d6f-1cfaa936313a

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

ID: 63a891c8-a46b-4d71-b238-5050140f737c

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

```
(define (=number? x y)
  (and (number? x)
       (number? y)
       (= x y)))
(define (=int? x y)
  (and (number? x)
       (number? y)
       (integer? x)
       (integer? y)
       (= x y)))
```

```
(define tower
  '(scheme-number rational real complex))

(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)))
```

```scheme
(define (install-real-package)
  ;; interface to rest of the system
  (define (tag x) (attach-tag 'real x))
  (put 'add '(real real)
       (lambda (x y) (tag (add x y))))
  (put 'sub '(real real)
       (lambda (x y) (tag (sub x y))))
  (put 'mul '(real real)
       (lambda (x y) (tag (mul x y))))
  (put 'div '(real real)
       (lambda (x y) (tag (div x y))))
  (put 'make 'real
       (lambda (r) (tag r)))
  'done)

(define (make-real r)
  ((get 'make 'real) r))
```

```scheme
<<op-table>>
<<type-table>>
<<sqrt>>
<<square>>
<<echo>>
<<smarter-type-tagging>>
<<attach-tag-fixed>> ;; time travel due to numer/denom fixes
<<apply-generic-coerce-less-naive>>
<<generic-arithmetic>>
<<scheme-number-package>>
<<real-package>>
<<equ?>>
<<eqzero?>>
<<real-package>>
<<basic-tower-coercions>>

(install-rational-package)
(install-rectangular-package)
(install-polar-package)
(install-complex-package)
(install-scheme-number-package)
(install-real-package)
(install-equ?)
(install-=zero?)
<<gen-math-tests>>

(let recurse ((x 42))
  (let ((x2 (raise x)))
    (if (eqv? #f x2)
```

```
        (echo "Done")
        (begin (echo x2)
               (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`

ID: 83cb8c70-5e8d-4462-b9e2-90859e74437e

3.96.1 Question

ID: f9313bf5-f209-4100-9284-7ef68e4977fc

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

ID: d0b5e765-1af3-403e-8209-b05550e178da

First some (unnecessarily general) ways to find items in a list.

```
(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)))

```

```
;; My first definition for max-index was this under the
;; assumption it would be faster, because the former
;; makes variables for each fold. But they benchmarked
;; identically.
;(define (max-index lst)
;  (define index car)
;  (define max-i cadr)
;  (define max-value caddr)
;  (define (folder x result)
;    (cons (1+ (index result))
;          (if (> x (max-value result))
;              (list (1+ (index result))
;                    x)
;              (list (max-i result)
;                    (max-value result)))))
;
;  (max-i (fold folder (list -1 0 0)
;               lst)))
```

The new apply-generic and helper functions.

```
;; type, type -> bool
;; Return true if type a is the lowest, false otherwise.
(define (is-earlier-in-tower? a b)
  (let ((result (list-indices tower a b)))
    (if (any (lambda(x) (eq? x #f)) result)
        (error "type missing from tower!" result)
        (apply < result))))
;; types -> type
;; Take types to check on tower. Finds the
;; highest-ranked and returns it.
(define (latest-in-tower . args)
  (let* ((types (if (and (= 1 (length args))
                         (list? (car args)))
                    (car args)
                    args))
         (result (apply list-indices
                        tower types)))
    (if (any (lambda(x) (eq? x #f)) result)
        (error "type missing from tower!"
               "args" args "types" types "result" result)
        (list-ref types (max-index result)))))
```

```
(use-srfis '(1))
;; Version of apply-generic that tries raising
;; its arguments.
(define (apply-generic op . args)
  (define types (map type-tag args))
```

```
  (define (map-raise highest-type item)
    (if (equal? (type-tag item) highest-type)
        item
        (let ((raised (raise item)))
          (if (not raised)
              (error "map-raise: could not raise:" item highest-type)
              (map-raise highest-type raised)))))

  (let ((proc (get op types)))
    (if proc ; is this already doable?
        (apply proc (map contents args))
                                        ; if not, try type coercion

        (let* ((highest-type (apply latest-in-tower types))
               (map-raise-prime (lambda(item)(map-raise highest-type item)))
               (raised (map map-raise-prime args)))
          (if (< (length args) 3)
              ;; short args
              (apply (get op (map type-tag raised))
                     (map contents raised))
              ;; long args
              (reduce (lambda(elem previous)
                        (apply-generic op previous elem))
                      "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:

```
(define (attach-tag type-tag contents)
  (if (eqv? type-tag 'scheme-number)
      contents
      (cons type-tag contents)))
```

```
(load "mattcheck2.scm")
<<op-table>>
<<type-table>>
<<sqrt>>
<<square>>
<<echo>>
<<smarter-type-tagging>>
<<attach-tag-fixed>>
<<list-indices>>
<<tower-predicates>>
<<apply-generic-raise>>
<<generic-arithmetic>>
<<scheme-number-package>>
<<real-package>>
```

```
<<equ?>>
<<eqzero?>>
<<real-package>>
<<basic-tower-coercions>>

(install-rational-package)
(install-rectangular-package)
(install-polar-package)
(install-complex-package)
(install-scheme-number-package)
(install-real-package)
(install-equ?)
(install-=zero?)
<<gen-math-tests>>

(mattcheck "is-earlier-in-tower? true"
           (is-earlier-in-tower? 'scheme-number 'real))
(mattcheck "is-earlier-in-tower? false"
           #f (is-earlier-in-tower? 'complex 'scheme-number))
(mattcheck "max-index"
           (max-index (list 1 3 5 4 2 0))
           2)
(mattcheck "latest-in-tower"
           (apply latest-in-tower tower)
           'complex)
(mattcheck "latest-in-tower reverse"
           (apply latest-in-tower (reverse tower))
           'complex)
(mattcheck "latest-in-tower 2 args"
           (apply latest-in-tower '(scheme-number real))
           'real)
(mattcheck "latest-in-tower 1 arg"
           (apply latest-in-tower '(real))
           'real)
(mattcheck "raise success"
           (raise 78)
           (make-rational 78 1))
(mattcheck "raise fail"
           (raise (make-complex-from-mag-ang 78 64))
           #f)
(mattcheck "apply-generic 1"
           (add 1 (make-rational 3 5))
           (make-rational 8 5))
(mattcheck "apply-generic 2"
           (add (make-real 2)
                (make-complex-from-mag-ang 3 5))
           (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`.

```
(add (make-real 2)
     (make-complex-from-mag-ang 3 5))
;; eventually reaches...
(add-complex (rectangular (real . 2) . 0)
             (polar 3 . 5))
;; 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`

ID: 88d8ef57-42ba-4e4c-b810-b40cbc67a912

3.97.1 Question

ID: c9389dd5-56c5-44d0-8869-dafbd36e54ba

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 `apply-generic` from Exercise 2.84 so that it "simplifies" its answers.

3.97.2 Answer

ID: fc350f55-cb72-435a-a521-5d64f3fa45d6

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
```

[12] A real number can be projected to an integer using the `round` primitive, which returns the closest integer to its argument.

```
          (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)))
(put 'project 'complex
     (lambda(num)
       (make-real
        ((get 'real-part '(complex)) num))))
```

```
(use-srfis '(1))
;; Version of apply-generic that tries to drop its type
;; when returning
(define (apply-generic op . args)
  (define types (map type-tag args))
  (define (map-raise highest-type item)
    (if (equal? (type-tag item) highest-type)
        item
        (let ((raised (raise item)))
          (if (not raised)
              (error ”MAP-RAISE: could not raise” item highest-type)
              (map-raise highest-type raised)))))

  (drop
   (let ((proc (get op types)))
     (if proc ; is this already doable?
         (apply proc (map contents args))
                                    ; if not, try type coercion

         (let* ((highest-type (apply latest-in-tower types))
                (map-raise-prime (lambda(item)(map-raise highest-type
                ↪  item)))
                (raised (map map-raise-prime args)))
```

```
           (if (< (length raised) 3)
               ;; short args
               (let ((f (get op (map type-tag raised))))
                 (if (eqv? #f f)
                     (error ⌟
                     ↪   "APPLY-GENERIC: could not find procedure for types"⌟
                     ↪    (map type-tag raised)
                     ↪   raised)
                     (apply f (map contents raised))))
               ;; long args
               (reduce (lambda(elem previous)
                         (apply-generic op previous elem))
                       "ERROR" raised)))))))
```

```
(load "mattcheck2.scm")
<<op-table>>
<<type-table>>
<<sqrt>>
<<square>>
<<echo>>
<<smarter-type-tagging>>
<<attach-tag-fixed>>
<<list-indices>>
<<tower-predicates>>
<<drop-project>>
<<apply-generic-drop>>
<<generic-arithmetic>>
<<scheme-number-package>>
<<real-package>>
<<equ?>>
<<eqzero?>>
<<real-package>>
<<basic-tower-coercions>>

(install-rational-package)
(install-rectangular-package)
(install-polar-package)
(install-complex-package)
(install-scheme-number-package)
(install-real-package)
(install-equ?)
(install-=zero?)
<<gen-math-tests>>

(mattcheck "raise success"
           (raise 78)
           (make-rational 78 1))
(mattcheck "raise fail"
```

```
             (raise (make-complex-from-mag-ang 78 64))
             #f)
  (mattcheck "apply-generic 1"
             (add 1 (make-rational 3 5))
             (make-rational 8 5))
  (mattcheck "apply-generic 2"
             (add (make-real 2)
                  (make-complex-from-mag-ang 3 5))
             (make-complex-from-real-imag 2.850986556389679
             ↪  -2.8767728239894153))
  (mattcheck "drop 1"
             (drop (make-complex-from-real-imag 5 0))
             5)
  (mattcheck "drop 2"
             (drop (make-real 5))
             5)
  (mattcheck "drop 3"
             (drop (make-rational 5 1))
             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

ID: 6f4359f5-e417-4d70-b75d-be454205cc69

3.98.1 Question

ID: 855bd300-42b2-44a7-bbb6-6aae4c0c813e

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

ID: fa40dd78-50ab-4eb3-8637-7f4286dbc7d6

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.

```
(define (add x y) (apply-generic 'add x y))
(define (sub x y) (apply-generic 'sub x y))
(define (mul x y) (apply-generic 'mul x y))
(define (div x y) (apply-generic 'div x y))
(define (real-part z) (apply-generic 'real-part z))
(define (imag-part z) (apply-generic 'imag-part z))
(define (magnitude z) (apply-generic 'magnitude z))
(define (angle z) (apply-generic 'angle z))
;; Now we begin to change code for support
;; These will be named slightly different so the primitive
;; functions aren't shadowed
(define (square x) (mul x 2))
(define (sine x) (apply-generic 'sine x)) ;; (sin)
(define (cosine x) (apply-generic 'cosine x)); (cos)
(define (square-root x) (apply-generic 'square-root x));(sqrt)
(define (arctan x y) (apply-generic 'arctan x y));(atan)

```

```
(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)
       (lambda (x y) (tag (/ x y))))
  (put 'sine '(scheme-number)
       (lambda (x) (tag (sin x))))
  (put 'cosine '(scheme-number)
       (lambda (x) (tag (cos x))))
  (put 'square-root '(scheme-number)
       (lambda (x) (tag (sqrt x))))
  (put 'arctan '(scheme-number scheme-number)
       (lambda (x y) (tag (atan x y))))
  (put 'make 'scheme-number (lambda (x) (tag x)))
  'done)

(define (install-rational-package)
  ;; internal procedures
  (define (numer x) (car x))
  (define (denom x) (cdr x))
  (define (make-rat n d)
    (if (or (inexact? n)
            (inexact? d))
        (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)
  (put 'denom '(rational) denom)
  (put 'sine '(rational)
       (lambda(r) ((get 'sine '(scheme-number))
                   (/ (numer r) (denom r)))))
  (put 'cosine '(rational)
       (lambda(r) ((get 'cosine '(scheme-number))
                   (/ (numer r) (denom r)))))
  (put 'square-root '(rational)
       (lambda(r) ((get 'square-root '(scheme-number))
                   (/ (numer r) (denom r)))))
  (put 'arctan '(rational)
       (lambda(r s) ((get 'arctan '(scheme-number))
                     (/ (numer r) (denom r)) (/ (numer s) (denom s)))))
  '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))
  (define (make-from-real-imag x y) (cons x y))
  (define (magnitude z)
    (square-root (add (square (real-part z))
                      (square (imag-part z)))))
  (define (angle z)
    (arctan (imag-part z) (real-part z)))
```

```
  (define (make-from-mag-ang r a)
    (cons (mul r (cosine a)) (mul r (sine a))))
  ;; interface to the rest of the system
  (define (tag x) (attach-tag 'rectangular x))
  (put 'real-part '(rectangular) real-part)
  (put 'imag-part '(rectangular) imag-part)
  (put 'magnitude '(rectangular) magnitude)
  (put 'angle '(rectangular) angle)
  (put 'make-from-real-imag 'rectangular
       (lambda (x y) (tag (make-from-real-imag x y))))
  (put 'make-from-mag-ang 'rectangular
       (lambda (r a) (tag (make-from-mag-ang r a))))
  'done)

(define (install-polar-package)
  ;; internal procedures
  (define (magnitude z) (car z))
  (define (angle z) (cdr z))
  (define (make-from-mag-ang r a) (cons r a))
  (define (real-part z) (mul (magnitude z) (cosine (angle z))))
  (define (imag-part z) (mul (magnitude z) (sine (angle z))))
  (define (make-from-real-imag x y)
    (cons (sqrt (+ (square x) (square y)))
          (atan y x)))
  ;; interface to the rest of the system
  (define (tag x) (attach-tag 'polar x))
  (put 'real-part '(polar) real-part)
  (put 'imag-part '(polar) imag-part)
  (put 'magnitude '(polar) magnitude)
  (put 'angle '(polar) angle)
  (put 'make-from-real-imag 'polar
       (lambda (x y) (tag (make-from-real-imag x y))))
  (put 'make-from-mag-ang 'polar
       (lambda (r a) (tag (make-from-mag-ang r a))))
  'done)

(define (install-complex-package)
  (define (make-from-real-imag x y)
    ((get 'make-from-real-imag 'rectangular) x y))
  (define (make-from-mag-ang r a)
    ((get 'make-from-mag-ang 'polar) r a))
  ;; let's make these generic
  (define (add-complex z1 z2)
    (make-from-real-imag (add (real-part z1) (real-part z2))
                         (add (imag-part z1) (imag-part z2))))
  (define (sub-complex z1 z2)
    (make-from-real-imag (sub (real-part z1) (real-part z2))
                         (sub (imag-part z1) (imag-part z2))))
```

```scheme
  (define (mul-complex z1 z2)
    (make-from-mag-ang (mul (magnitude z1) (magnitude z2))
                       (add (angle z1) (angle z2))))
  (define (div-complex z1 z2)
    (make-from-mag-ang (div (magnitude z1) (magnitude z2))
                       (sub (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))

(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)
                             (apply-generic 'magnitude y))
                       (equ? (apply-generic 'angle x)
                             (apply-generic 'angle y))))
                 (else
                  (and (equ? (apply-generic 'real-part x)
                             (apply-generic 'real-part y))
                       (equ? (apply-generic 'imag-part x)
                             (apply-generic 'imag-part y))))))))
  (put 'equ? '(rational rational)
       (lambda(x y)
         (and (= ((get 'numer '(rational)) x)
                 ((get 'numer '(rational)) y))
              (= ((get 'denom '(rational)) x)
                 ((get 'denom '(rational)) y)))))
  (put 'equ? '(real real)
       (lambda(x y) (= x y))))
```

```
(load ”mattcheck2.scm”)
<<op-table>>
<<type-table>>
<<sqrt>>
```

```
<<echo>>
<<smarter-type-tagging>>
<<attach-tag-fixed>>
<<list-indices>>
<<tower-predicates>>
<<apply-generic-drop>>
<<generic-arithmetic-complex-parts>>
<<real-package>>
<<eqzero?>>
<<real-package>>
<<basic-tower-coercions>>

(install-rational-package)
(install-rectangular-package)
(install-polar-package)
(install-complex-package)
(install-scheme-number-package)
(install-real-package)
(install-equ?)
(install-=zero?)
<<gen-math-tests>>

(mattcheck "raise success"
           (raise 78)
           (make-rational 78 1))
(mattcheck "raise fail"
           (raise (make-complex-from-mag-ang 78 64))
           #f)
(mattcheck "apply-generic 1"
           (add 1 (make-rational 3 5))
           (make-rational 8 5))
(mattcheck "apply-generic 2"
           (add (make-real 2)
                (make-complex-from-mag-ang 3 5))
           (make-complex-from-real-imag 2.850986556389679
           ↪  -2.8767728239894153))
(mattcheck "drop 1"
           (drop (make-complex-from-real-imag 5 0))
           5)
(mattcheck "drop 2"
           (drop (make-real 5))
           5)
(mattcheck "drop 3"
           (drop (make-rational 5 1))
           5)
(mattcheck "complex with nested rational"
           (make-complex-from-real-imag 2 (make-rational 3 5))
```

```
            (cons 'complex (cons 'rectangular (cons 2 (cons 'rational
            ↪  (cons 3 5))))))
 (mattcheck "math on complex with nested rational"
            (add (make-complex-from-real-imag 2 (make-rational 3 5))
                 (make-complex-from-mag-ang (make-rational 3 5) 2))
            (cons 'complex (cons 'rectangular
                  (cons (cons 'rational (cons 8.751559490358574 5))
                        (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

ID: 29ffdd17-f543-432c-a51c-06d0154dafb5

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.

```
(define (install-polynomial-package)
  ;; internal procedures
  ;; representation of poly
  (define (make-poly variable term-list) (cons variable term-list))
  (define (variable p) (car p))
  (define (terms p) (cdr p)) ;; renamed from term-list since many procs
  ↪  happen
  (define (variable? x) (symbol? x))  ; to use "term-list" as an
  ↪  internal variable
  (define (same-variable? v1 v2)
    (and (variable? v1) (variable? v2) (eq? v1 v2)))
  ;; representation of terms and term lists
  (define (adjoin-term term term-list)
    (if (=zero? (coeff term))
        term-list
        (cons term term-list)))
  (define (the-empty-termlist) '())
  (define (first-term term-list) (car term-list))
  (define (rest-terms term-list) (cdr term-list))
  (define (empty-termlist? term-list) (null? term-list))
  (define (make-term order coeff) (list order coeff))
  (define (order term) (car term))
  (define (coeff term) (cadr term))
  (define (add-poly p1 p2)
    (if (same-variable? (variable p1) (variable p2))
```

```
        (make-poly (variable p1)
                   (add-terms (terms p1) (terms p2)))
        (error "Polys not in same var: ADD-POLY" (list p1 p2))))
  (define (mul-poly p1 p2)
    (if (same-variable? (variable p1) (variable p2))
        (make-poly (variable p1)
                   (mul-terms (terms p1) (terms p2)))
        (error "Polys not in same var: MUL-POLY" (list p1 p2))))
  (define (add-terms L1 L2)
    (cond ((empty-termlist? L1) L2)
          ((empty-termlist? L2) L1)
          (else
           (let ((t1 (first-term L1))
                 (t2 (first-term L2)))
             (cond ((> (order t1) (order t2))
                    (adjoin-term
                     t1 (add-terms (rest-terms L1) L2)))
                   ((< (order t1) (order t2))
                    (adjoin-term
                     t2 (add-terms L1 (rest-terms L2))))
                   (else
                    (adjoin-term
                     (make-term (order t1)
                                (add (coeff t1) (coeff t2)))
                     (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 (mul-term-by-all-terms t1 L)
    (if (empty-termlist? L)
        (the-empty-termlist)
        (let ((t2 (first-term L)))
          (adjoin-term
           (make-term (+ (order t1) (order t2))
                      (mul (coeff t1) (coeff t2)))
```

```scheme
            (mul-term-by-all-terms t1 (rest-terms L))))))
```

```scheme
  'done)

(define (make-polynomial var terms)
  ((get 'make 'polynomial) var terms))
```

3.100 Exercise 2.87: `=zero?` for polynomials

`ID: d930cb60-1e02-40df-a494-9b3a53d7bf17`

3.100.1 Question

`ID: 32f9ab8e-9e11-416b-97dd-88612f57c7d8`

Install `=zero?` for polynomials in the generic arithmetic package. This will allow `adjoin-term` to work for polynomials with coefficients that are themselves polynomials.

3.100.2 Answer

`ID: fd96d044-5030-4dc1-9b7f-d2eb1a80f403`

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))))))
```

```
(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?)
```

```
(load "mattcheck2.scm")
<<op-table>>
<<type-table>>
<<sqrt>>
<<echo>>
<<smarter-type-tagging>>
<<attach-tag-fixed>>
<<list-indices>>
<<tower-predicates>>
<<apply-generic-drop>>
<<generic-arithmetic-complex-parts>>
<<real-package>>
<<eqzero?>>
<<real-package>>
<<basic-tower-coercions>>

<<polynomials-package-top>>
<<mul-term-by-all-terms-mine>>
<<poly-terms-eqzero>>
<<polynomials-package-bottom>>

(install-rational-package)
(install-rectangular-package)
(install-polar-package)
(install-complex-package)
(install-scheme-number-package)
(install-real-package)
(install-equ?)
(install-=zero?)
(install-polynomial-package)
<<gen-math-tests>>

(mattcheck "raise success"
           (raise 78)
           (make-rational 78 1))
```

```
(mattcheck "raise fail"
           (raise (make-complex-from-mag-ang 78 64))
           #f)
(mattcheck "apply-generic 1"
           (add 1 (make-rational 3 5))
           (make-rational 8 5))
(mattcheck "apply-generic 2"
           (add (make-real 2)
                (make-complex-from-mag-ang 3 5))
           (make-complex-from-real-imag 2.850986556389679
           ↪  -2.8767728239894153))
(mattcheck "drop 1"
           (drop (make-complex-from-real-imag 5 0))
           5)
(mattcheck "drop 2"
           (drop (make-real 5))
           5)
(mattcheck "drop 3"
           (drop (make-rational 5 1))
           5)
(mattcheck "complex with nested rational"
           (make-complex-from-real-imag 2 (make-rational 3 5))
           (cons 'complex (cons 'rectangular (cons 2 (cons 'rational
           ↪  (cons 3 5))))))
(mattcheck "math on complex with nested rational"
           (add (make-complex-from-real-imag 2 (make-rational 3 5))
                (make-complex-from-mag-ang (make-rational 3 5) 2))
           (cons 'complex (cons 'rectangular
                                (cons (cons 'rational (cons
                                ↪  8.751559490358574 5))
                                      (cons 'rational (cons
                                      ↪  28.639461402385223 25))))))
(mattcheck "polynomial = zero"
           (=zero? (make-polynomial 'x '((0 0) (0 0)))))
(mattcheck "polynomial != zero" #f
           (=zero? (make-polynomial 'x '((0 0) (1 1)))))
(let ((p1 '((100 1)(2 2)(0 1)))
      (p2 (list (list 7 (make-polynomial 'x '((3 2)))))))
  (mattcheck "polynomial addition"
             (add (make-polynomial 'x p1)
                  (make-polynomial 'x p1))
             '(polynomial x (100 2)(2 4)(0 2)))
  (mattcheck "polynomial multiplication"
             (mul (make-polynomial 'x p1)
                  (make-polynomial 'x p1))
             '(polynomial x (200 1) (102 4) (100 2) (4 4) (2 4) (0 1)))
  (mattcheck "nested polynomial addition"
             (add (make-polynomial 'x p2)
```

```
                    (make-polynomial 'x p1))
               '(polynomial x (100 1) (7 (polynomial x (3 2))) (2 2) (0
               ↪  1)))
    (mattcheck "nested polynomial mult"
               (mul (make-polynomial 'x p2)
                    (make-polynomial 'x p1))
               '(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

ID: 4a4f3f20-e863-4637-bbcb-37072d4d0dba

3.101.1 Question

ID: 7183e9d9-de4c-418b-9810-122eb8ff5bd5

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

ID: 4d03f69b-46f7-400b-a40a-e55c14072281

```
(define (neg x)
  (apply-generic 'neg x))
(define (install-negation)
  (put 'neg '(scheme-number)
       (lambda(x) (- x)))
  (put 'neg '(rational)
       (lambda(R) (make-rational (neg ((get 'numer '(rational)) R))
                                  ((get 'denom '(rational)) R))))
  (put 'neg '(real) (get 'neg '(scheme-number)))
  (put 'neg '(rectangular)
       (lambda (C)
         (make-complex-from-real-imag (neg ((get 'real-part
         ↪  '(rectangular)) C))
                                      (neg ((get 'imag-part
                                      ↪  '(rectangular)) C)))))
  (put 'neg '(polar)
```

```
        (lambda (C)
          (make-complex-from-mag-ang (neg ((get 'magnitude '(polar)) C))
                                     ((get 'angle '(polar)) C))))
   (put 'neg '(complex)
        (lambda(C) (apply-generic 'neg C))))
```

```
      ;; inserted in install routine
      (define (neg-poly p)
      (make-poly (variable p)
                  (map (lambda(t)
                          (make-term (order t) (neg (coeff t))))
                      (terms p))))
      (put 'neg '(polynomial)
          (lambda (p1) (tag (neg-poly p1))))
      (define (sub-poly p1 p2)
      (if (same-variable? (variable p1) (variable p2))
          (add-poly p1 (neg-poly p2))
          (error "Polys not in same var: SUB-POLY" (list p1 p2))))
      (put 'sub '(polynomial polynomial)
          (lambda (p1 p2) (tag (sub-poly p1 p2))))
```

```
(load "mattcheck2.scm")
<<op-table>>
<<type-table>>
<<sqrt>>
<<echo>>
<<smarter-type-tagging>>
<<attach-tag-fixed>>
<<list-indices>>
<<tower-predicates>>
<<apply-generic-drop>>
<<generic-arithmetic-complex-parts>>
<<real-package>>
<<eqzero?>>
<<real-package>>
<<basic-tower-coercions>>

<<polynomials-package-top>>
<<mul-term-by-all-terms-mine>>
<<poly-terms-eqzero>>
<<polynomial-subtraction>>
<<polynomials-package-bottom>>

<<generic-negation>>

(install-rational-package)
(install-rectangular-package)
```

```
(install-polar-package)
(install-complex-package)
(install-scheme-number-package)
(install-real-package)
(install-equ?)
(install-=zero?)
(install-polynomial-package)
(install-negation)

<<gen-math-tests>>

(mattcheck "raise success"
           (raise 78)
           (make-rational 78 1))
(mattcheck "raise fail"
           (raise (make-complex-from-mag-ang 78 64))
           #f)
(mattcheck "apply-generic 1"
           (add 1 (make-rational 3 5))
           (make-rational 8 5))
(mattcheck "apply-generic 2"
           (add (make-real 2)
                (make-complex-from-mag-ang 3 5))
           (make-complex-from-real-imag 2.850986556389679
           ↪   -2.8767728239894153))
(mattcheck "drop 1"
           (drop (make-complex-from-real-imag 5 0))
           5)
(mattcheck "drop 2"
           (drop (make-real 5))
           5)
(mattcheck "drop 3"
           (drop (make-rational 5 1))
           5)
(mattcheck "complex with nested rational"
           (make-complex-from-real-imag 2 (make-rational 3 5))
           (cons 'complex (cons 'rectangular (cons 2 (cons 'rational
           ↪   (cons 3 5))))))
(mattcheck "math on complex with nested rational"
           (add (make-complex-from-real-imag 2 (make-rational 3 5))
                (make-complex-from-mag-ang (make-rational 3 5) 2))
           (cons 'complex (cons 'rectangular
                                (cons (cons 'rational (cons
                                ↪   8.751559490358574 5))
                                      (cons 'rational (cons
                                      ↪   28.639461402385223 25))))))
(mattcheck "polynomial = zero"
           (=zero? (make-polynomial 'x '((0 0) (0 0)))))
```

```
(mattcheck "polynomial != zero" #f
           (=zero? (make-polynomial 'x '((0 0) (1 1)))))
(let ((p1 '((100 1)(2 2)(0 1)))
      (p2 (list (list 7 (make-polynomial 'x '((3 2)))))))
  (mattcheck "polynomial addition"
             (add (make-polynomial 'x p1)
                  (make-polynomial 'x p1))
             '(polynomial x (100 2)(2 4)(0 2)))
  (mattcheck "polynomial multiplication"
             (mul (make-polynomial 'x p1)
                  (make-polynomial 'x p1))
             '(polynomial x (200 1) (102 4) (100 2) (4 4) (2 4) (0 1)))
  (mattcheck "nested polynomial addition"
             (add (make-polynomial 'x p2)
                  (make-polynomial 'x p1))
             '(polynomial x (100 1) (7 (polynomial x (3 2))) (2 2) (0
             ↪  1)))
  (mattcheck "nested polynomial mult"
             (mul (make-polynomial 'x p2)
                  (make-polynomial 'x p1))
             '(polynomial x (107 (polynomial x (3 2))) (9 (polynomial x
             ↪  (3 4))) (7 (polynomial x (3 2))))))
(mattcheck "negation scheme-number"
           (neg 5) -5)
(mattcheck "negation rational"
           (neg (make-rational 1 5))
           (make-rational -1 5))
(mattcheck "negation rectangular complex"
           (neg (make-complex-from-real-imag 3 7))
           (make-complex-from-real-imag -3 -7))
(mattcheck "negation polar complex"
           (neg (make-complex-from-mag-ang 3 7))
           (make-complex-from-mag-ang -3 7))

(let ((p1 '((100 1)(2 2)(0 1)))
      (p2 (list (list 7 (make-polynomial 'x '((3 2))))))
      (p3 '((100 1)(2 1)(0 3)))
      (p4 '((100 -1)(2 -2)(0 -1))))
  (mattcheck "negation polynomials"
             (neg (make-polynomial 'x p1))
             (make-polynomial 'x p4))
  (mattcheck "polynomial subtraction"
             (sub (make-polynomial 'x p1)
                  (make-polynomial 'x p3))
             '(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

ID: 2339a4e3-dac4-481c-bc64-02351946664c

3.102.1 Question

ID: a0d98187-5b82-43d6-b1ee-7b9467e612f3

Define procedures that implement the term-list representation described above as appropriate for dense polynomials.

3.102.2 Answer

ID: 5ebf9444-331d-403b-b080-4ad16a6e720b

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.

```scheme
(define (sparse-termlist? term-list)
  (let ((t (car term-list)))
    (cond ((and (list? t)
                (number? (first t))
                (number? (second t))) #t)
          ((number? t) #f)
          (else (error "badly formed term: SPARSE-TERMLIST?" t
          ↪  term-list)))))
(define (make-sparse-term order coeff)
  (list order coeff))
(define (dense-first-term term-list)
  (make-term (car term-list)
             (1- (length 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)))
```

3.103 Exercise 2.90: dense polynomials integrated into the system

ID: d313a504-3248-4855-92b5-e0bfb2461517

3.103.1 Question

ID: fcc174bc-c27a-47a4-abe5-82ef200c1e85

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

ID: 4a1b260a-bddb-4377-9979-51611ddb83ad

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.

```
(use-srfis '(1))
(define (install-polynomial-package)
  ;; internal procedures
  ;; representation of poly
  (define (make-poly variable term-list)
    (cons variable
          (if (empty-termlist? term-list)
              term-list
```

```scheme
            (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)
                (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-first-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)
                    (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)
;;                                    (adjoin-term-dense term
;;                                                       (rest-terms term-list)))))))

(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 (the-empty-termlist) '())
(define (rest-terms term-list) (cdr term-list))
(define (empty-termlist? term-list) (null? term-list))
```

```
  (define (order term) (first term))
  (define (coeff term) (second term))
  (define (add-poly p1 p2)
    (if (same-variable? (variable p1) (variable p2))
        (make-poly (variable p1)
                   (add-terms (terms p1) (terms p2)))
        (error "Polys not in same var: ADD-POLY" (list p1 p2))))
  (define (mul-poly p1 p2)
    (if (same-variable? (variable p1) (variable p2))
        (make-poly (variable p1)
                   (mul-terms (terms p1) (terms p2)))
        (error "Polys not in same var: MUL-POLY" (list p1 p2))))
  (define (add-terms L1 L2)
    (cond ((empty-termlist? L1) L2)
          ((empty-termlist? L2) L1)
          (else
           (let ((t1 (first-term L1))
                 (t2 (first-term L2)))
             (cond ((> (order t1) (order t2))
                    (adjoin-term
                     t1 (add-terms (rest-terms L1) L2)))
                   ((< (order t1) (order t2))
                    (adjoin-term
                     t2 (add-terms L1 (rest-terms L2))))
                   (else
                    (adjoin-term
                     (make-sparse-term (order t1)
                                       (add (coeff t1) (coeff t2)))
                     (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 (if (sparse-termlist? TL)
                       (lambda(t)
                         (make-sparse-term (order t) (neg (coeff t))))
                       (lambda(t)
                         (neg (coeff t))))))
      (map negterm TL)))
  (define (sub-terms T1 T2)
    (add-terms T1 (neg-terms T2)))
  (define (neg-poly p)
    (make-poly (variable p)
               (neg-terms (terms p))))
  (put 'neg '(polynomial)
       (lambda (p1) (tag (neg-poly p1))))
  (define (sub-poly p1 p2)
    (add-poly p1 (neg-poly p2)))
  (put 'sub '(polynomial polynomial)
       (lambda (p1 p2) (tag (sub-poly p1 p2))))
```

```
  'done)
(define (make-polynomial var terms)
  ((get 'make 'polynomial) var terms))
```

```
(load "mattcheck2.scm")
<<op-table>>
<<type-table>>
<<sqrt>>
<<echo>>
<<smarter-type-tagging>>
<<attach-tag-fixed>>
<<list-indices>>
<<tower-predicates>>
<<apply-generic-drop>>
<<generic-arithmetic-complex-parts>>
<<real-package>>
<<eqzero?>>
<<real-package>>
<<basic-tower-coercions>>

<<polynomial-sparse-dense-top>>
<<polynomial-sparse-dense-bottom>>

<<generic-negation>>

(install-rational-package)
(install-rectangular-package)
(install-polar-package)
(install-complex-package)
(install-scheme-number-package)
(install-real-package)
(install-equ?)
(install-=zero?)
(install-polynomial-package)
(install-negation)

<<gen-math-tests>>

(mattcheck "raise success"
           (raise 78)
           (make-rational 78 1))
(mattcheck "raise fail"
           (raise (make-complex-from-mag-ang 78 64))
           #f)
(mattcheck "apply-generic 1"
           (add 1 (make-rational 3 5))
           (make-rational 8 5))
(mattcheck "apply-generic 2"
           (add (make-real 2)
                (make-complex-from-mag-ang 3 5))
           (make-complex-from-real-imag 2.850986556389679
           ↪  -2.8767728239894153))
```

```
(mattcheck "drop 1"
           (drop (make-complex-from-real-imag 5 0))
           5)
(mattcheck "drop 2"
           (drop (make-real 5))
           5)
(mattcheck "drop 3"
           (drop (make-rational 5 1))
           5)
(mattcheck "complex with nested rational"
           (make-complex-from-real-imag 2 (make-rational 3 5))
           (cons 'complex (cons 'rectangular (cons 2 (cons 'rational (cons 3 5))))))
(mattcheck "math on complex with nested rational"
           (add (make-complex-from-real-imag 2 (make-rational 3 5))
                (make-complex-from-mag-ang (make-rational 3 5) 2))
           (cons 'complex (cons 'rectangular
                                 (cons (cons 'rational (cons 8.751559490358574 5))
                                       (cons 'rational (cons 28.639461402385223 25))))))
(mattcheck "polynomial = zero"
           (=zero? (make-polynomial 'x '((0 0)))))
(mattcheck "polynomial != zero" #f
           (=zero? (make-polynomial 'x '((1 1) (0 1)))))

(let ((p1 '((100 1)(2 2)(0 1)))
      (p2 (list (list 7 (make-polynomial 'x '((3 2))))))
      (p3 '((100 1)(2 1)(0 3)))
      (p4 '((100 -1)(2 -2)(0 -1)))
      (p5 '(8 2 5 -3 -7)))
  (mattcheck "polynomial addition"
             (add (make-polynomial 'x p1)
                  (make-polynomial 'x p1))
             '(polynomial x (100 2)(2 4)(0 2)))
  (mattcheck "polynomial multiplication"
             (mul (make-polynomial 'x p1)
                  (make-polynomial 'x p1))
             '(polynomial x (200 1) (102 4) (100 2) (4 4) (2 4) (0 1)))
  (mattcheck "nested polynomial addition"
             (add (make-polynomial 'x p2)
                  (make-polynomial 'x p1))
             '(polynomial x (100 1) (7 (polynomial x (3 2))) (2 2) (0 1)))
  (mattcheck "nested polynomial mult"
             (mul (make-polynomial 'x p2)
                  (make-polynomial 'x p1))
```

```
              '(polynomial x (107 (polynomial x (3 2))) (9 (polynomial x
              ↪  (3 4))) (7 (polynomial x (3 2)))))
    (mattcheck "negation scheme-number"
              (neg 5) -5)
    (mattcheck "negation rational"
              (neg (make-rational 1 5))
              (make-rational -1 5))
    (mattcheck "negation rectangular complex"
              (neg (make-complex-from-real-imag 3 7))
              (make-complex-from-real-imag -3 -7))
    (mattcheck "negation polar complex"
              (neg (make-complex-from-mag-ang 3 7))
              (make-complex-from-mag-ang -3 7))

    (mattcheck "negation polynomials"
              (neg (make-polynomial 'x p1))
              (make-polynomial 'x p4))
    (mattcheck "polynomial subtraction"
              (sub (make-polynomial 'x p1)
                   (make-polynomial 'x p3))
              '(polynomial x 1 0 -2)))
```

```
(etc...)
SUCCEED at negation polynomials
SUCCEED at polynomial subtraction
```

3.104 Exercise 2.91: division of polynomials

ID: 1b96bdbf-924e-4fd5-a614-b8bde734c694

3.104.1 Question

ID: affbc03d-488d-457b-8c85-b13bd76bb601

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-terms` 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.

```
(define (div-terms L1 L2)
  (if (empty-termlist? L1)
      (list (the-empty-termlist) (the-empty-termlist))
      (let ((t1 (first-term L1))
            (t2 (first-term L2)))
        (if (> (order t2) (order t1))
            (list (the-empty-termlist) L1)
            (let ((new-c (div (coeff t1) (coeff t2)))
                  (new-o (- (order t1) (order t2))))
              (let ((rest-of-result
                     ;;<compute rest of result recursively>
                     ))
                ;;<form complete result>
                ))))))
```

3.104.2 Answer

ID: cc69778d-81de-4551-9734-fd0c56437f3a

```
(define (div-terms L1 L2)
  ;; termlist, termlist -> list of two termlists
  ;; returns quotient terms and remainder terms
  (if (empty-termlist? L1)
      (list (the-empty-termlist) (the-empty-termlist))
      (let ((t1 (first-term L1))
            (t2 (first-term L2)))
        (if (> (order t2) (order t1))
            (list (the-empty-termlist) L1)
            (let* ((new-term (make-sparse-term (- (order t1) (order t2))
                                               (div (coeff t1) (coeff t2))))
                   (rest-of-result
                    (div-terms (sub-terms L1
                                          (mul-terms (list new-term)
                                                     L2))
                               L2)))
              (list (adjoin-term new-term
                                 (car rest-of-result))
                    (cadr rest-of-result)))))))
```

```scheme
(define (div-poly p1 p2)
  ;; poly, poly -> list of 2 polys
  ;; returns quotient poly and remainder poly
  (if (same-variable? (variable p1) (variable p2))
      (let* ((result (div-terms (terms p1) (terms p2)))
             (quotient (car result))
             (remainder (cadr result)))
        (list (make-poly (variable p1) quotient)
              (make-poly (variable p1) remainder)))
      (error "Polys not in same var: DIV-POLY" (list p1 p2))))
(put 'div '(polynomial polynomial)
     (lambda (p1 p2)
       (let* ((result (div-poly p1 p2))
              (quotient (car result))
              (remainder (cadr result)))
         (list (tag quotient)
               (tag remainder)))))
```

```scheme
(load "mattcheck2.scm")
<<op-table>>
<<type-table>>
<<sqrt>>
<<echo>>
<<smarter-type-tagging>>
<<attach-tag-fixed>>
<<list-indices>>
<<tower-predicates>>
<<apply-generic-drop>>
<<generic-arithmetic-complex-parts>>
<<real-package>>
<<eqzero?>>
<<real-package>>
<<basic-tower-coercions>>

<<polynomial-sparse-dense-top>>
<<polynomial-division>>
<<polynomial-sparse-dense-bottom>>

<<generic-negation>>

(install-rational-package)
(install-rectangular-package)
(install-polar-package)
(install-complex-package)
(install-scheme-number-package)
(install-real-package)
(install-equ?)
(install-=zero?)
```

```
(install-polynomial-package)
(install-negation)

<<gen-math-tests>>

(mattcheck "raise success"
           (raise 78)
           (make-rational 78 1))
(mattcheck "raise fail"
           (raise (make-complex-from-mag-ang 78 64))
           #f)
(mattcheck "apply-generic 1"
           (add 1 (make-rational 3 5))
           (make-rational 8 5))
(mattcheck "apply-generic 2"
           (add (make-real 2)
                (make-complex-from-mag-ang 3 5))
           (make-complex-from-real-imag 2.850986556389679 -2.8767728239894153))
(mattcheck "drop 1"
           (drop (make-complex-from-real-imag 5 0))
           5)
(mattcheck "drop 2"
           (drop (make-real 5))
           5)
(mattcheck "drop 3"
           (drop (make-rational 5 1))
           5)
(mattcheck "complex with nested rational"
           (make-complex-from-real-imag 2 (make-rational 3 5))
           (cons 'complex (cons 'rectangular (cons 2 (cons 'rational (cons 3 5))))))
(mattcheck "math on complex with nested rational"
           (add (make-complex-from-real-imag 2 (make-rational 3 5))
                (make-complex-from-mag-ang (make-rational 3 5) 2))
           (cons 'complex (cons 'rectangular
                                (cons (cons 'rational (cons 8.751559490358574 5))
                                      (cons 'rational (cons 28.639461402385223 25))))))
(mattcheck "polynomial = zero"
           (=zero? (make-polynomial 'x '((0 0)))))
(mattcheck "polynomial != zero" #f
           (=zero? (make-polynomial 'x '((1 1)(0 1)))))

(let ((p1 '((100 1)(2 2)(0 1)))
      (p2 (list (list 7 (make-polynomial 'x '((3 2))))))
      (p3 '((100 1)(2 1)(0 3)))
```

```
        (p4 '((100 -1)(2 -2)(0 -1)))
        (p5 '(8 2 5 -3 -7))
        (p7 '((1 1)(0 1)))
        (p6 '((3 1)(0 1)))
        (p66 '(1 0 0 1))
        (p77 '(1 1)))
    (mattcheck "polynomial addition"
               (add (make-polynomial 'x p1)
                    (make-polynomial 'x p1))
               '(polynomial x (100 2)(2 4)(0 2)))
    (mattcheck "polynomial multiplication"
               (mul (make-polynomial 'x p1)
                    (make-polynomial 'x p1))
               '(polynomial x (200 1) (102 4) (100 2) (4 4) (2 4) (0 1)))
    (mattcheck "nested polynomial addition"
               (add (make-polynomial 'x p2)
                    (make-polynomial 'x p1))
               '(polynomial x (100 1) (7 (polynomial x (3 2))) (2 2) (0 1)))
    (mattcheck "nested polynomial mult"
               (mul (make-polynomial 'x p2)
                    (make-polynomial 'x p1))
               '(polynomial x (107 (polynomial x (3 2))) (9 (polynomial x (3 4))) (7 (polynomial x (3 2)))))
    (mattcheck "negation scheme-number"
               (neg 5) -5)
    (mattcheck "negation rational"
               (neg (make-rational 1 5))
               (make-rational -1 5))
    (mattcheck "negation rectangular complex"
               (neg (make-complex-from-real-imag 3 7))
               (make-complex-from-real-imag -3 -7))
    (mattcheck "negation polar complex"
               (neg (make-complex-from-mag-ang 3 7))
               (make-complex-from-mag-ang -3 7))

    (mattcheck "negation polynomials"
               (neg (make-polynomial 'x p1))
               (make-polynomial 'x p4))
    (mattcheck "polynomial subtraction"
               (sub (make-polynomial 'x p1)
                    (make-polynomial 'x p3))
               '(polynomial x 1 0 -2))
    (mattcheck "polynomial division 1"
               (div (make-polynomial 'x p6)
                    (make-polynomial 'x p7))
               '((polynomial x 1 -1 1) (polynomial x)))
    (mattcheck "polynomial division 2"
```

```
               (div (make-polynomial 'x p7)
                    (make-polynomial 'x p6))
               '((polynomial x) (polynomial x 1 1)))
    (mattcheck "polynomial division 3"
               (div (make-polynomial 'x p66)
                    (make-polynomial 'x p77))
               '((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

ID: 42444d8c-8e71-4d6e-8bb0-b88a65777808

3.105.1 Question

ID: 7960a7b0-063b-40b6-b708-bd1c8e209c3b

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

ID: 7b686824-168f-4264-9ee9-d936202ac19f

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^{y^z} and y with w^{z^y}.

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 it'll get keyed to the variable of the poly, and if it's not it'll be keyed to the symbol `'ZEROTH`.

```
(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))))))
```

```
(use-srfis '(1))
(define (mem-index item ll)
  (list-index (lambda(x)(equal? item x))
              ll))
```

```
(use-srfis '(1))
(define (install-polynomial-package) 'no)
;; internal procedures
;; representation of poly
```

```scheme
(define (make-poly variable term-list)
  (cons variable
        (convert-termlist-if-reasonable 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)))
(define (is-poly? list)
  (eqv? 'polynomial (type-tag list)))

;; representation of terms and term lists
(define (the-empty-termlist) '())
(define (rest-terms term-list) (cdr term-list))
(define (empty-termlist? term-list) (null? term-list))
(define (order term) (first term))
(define (coeff term) (second term))
(define (make-sparse-term order coeff)
  (list order coeff))
(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)))) #t)
          ((or (number? t) ;; looks dense
               (pair? t)) #f) ;; or dense with a nested poly
          (else (error "badly formed term: SPARSE-TERMLIST?" t
          ↪  term-list)))))
;; term and termlist procedures
(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)
                (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 (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 (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 (dense-termlist-order term-list)
  (1- (length term-list)))
(define (dense-first-term term-list)
  (make-sparse-term (dense-termlist-order term-list)
```

```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)
  ;; 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
                                                                  (sparse-first-term term-list))))
                      (cons result
                            (rest-terms 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 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
```

```scheme
           (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-first-term term-list))))
                 (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-list)))))))

(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)
                           (merge-terms t1 (rest-terms t2))))))
                                        ; ;; this second version is way simpler and way slower.
                                        ; (define (merge-terms-abstracted t1 t2)
                                        ;   (if (empty-termlist? t2)
                                        ;       t1
```

```
                                        ;       (merge-terms-abstracted (adjoin-term (first-term t2) t1)
                                        ;      (rest-terms t2))))
(define (map-terms f tl)
  ;; f is called (f order coeff)
  (let iter ((result '())
             (term-list tl))
    (if (empty-termlist? term-list)
        (reverse result)
        (iter (cons (let ((t (first-term term-list)))
                      (f (order t) (coeff t)))
                    result)
              (rest-terms term-list)))))
(define (map2-terms f tl1 tl2)
  (let iter ((result '())
             (term-list1 (reverse tl1))
             (term-list2 (reverse tl2)))
    (if (or (empty-termlist? term-list1)
            (empty-termlist? term-list2))
        (reverse result)
        (iter (cons (let ((t1 (first-term term-list))
                          (t2 (first-term term-list)))
                      (f (order t1) (coeff t1)
                         (order t2) (coeff t2)))
                    result)
              (rest-terms term-list1)
              (rest-terms term-list2)))))
(define (equ?-terms t1 t2)
  (and (= (order t1)
          (order t2))
       (equ? (coeff t1)
             (coeff t2))))
(define (sort-sparse-terms term-list)
  (sort term-list (lambda(t1 t2)
                    (> (order t1)
                       (order t2)))))
(define (add-sparse-term-to-sparse-term t1 t2)
  (make-sparse-term (order t1)
                    (add (coeff t1) (coeff t2))))
(define (add-terms L1 L2)
  (cond ((empty-termlist? L1) L2)
        ((empty-termlist? L2) L1)
        (else
         (let ((t1 (first-term L1))
               (t2 (first-term L2)))
           (cond ((> (order t1) (order t2))
```

```
                  (adjoin-term
                   t1 (add-terms (rest-terms L1) L2)))
                 ((< (order t1) (order t2))
                  (adjoin-term
                   t2 (add-terms L1 (rest-terms L2))))
                 (else
                  (adjoin-term
                   (add-sparse-term-to-sparse-term t1 t2)
                   (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))))
(define (terms-=zero? L)
  (or (empty-termlist? L)
      (and (=zero? (coeff (first-term L)))
           (terms-=zero? (rest-terms L)))))
(define (neg-terms TL)
  (let ((negterm (if (sparse-termlist? TL)
                     (lambda(t)
                       (make-sparse-term (order t) (neg (coeff t))))
                     (lambda(t)
                       (neg (coeff t))))))
    (map negterm TL)))
(define (sub-terms T1 T2)
  (add-terms T1 (neg-terms T2)))
(define (div-terms L1 L2)
  ;; termlist, termlist -> list of two termlists
  ;; returns quotient terms and remainder terms
  (if (empty-termlist? L1)
      (list (the-empty-termlist) (the-empty-termlist))
      (let ((t1 (first-term L1))
            (t2 (first-term L2)))
        (if (> (order t2) (order t1))
            (list (the-empty-termlist) L1)
            (let* ((new-term (make-sparse-term (- (order t1) (order t2))
                                               (div (coeff t1) (coeff
                                               ↪  t2))))
                   (rest-of-result
                    (div-terms (sub-terms L1
                                          (mul-terms (list new-term)
                                                     L2))
                               L2)))
              (list (adjoin-term new-term
                                 (car rest-of-result))
                    (cadr rest-of-result)))))))
```

```scheme
;; polynomial procedures
(define (trivial-poly? poly)
  (and (= 1 (length (terms poly)))
       (= 0 (order (first-term (terms poly))))))
;;; TODO: get working once cross-variable comparisons can be done
(put 'project 'polynomial
     (lambda (p)
       (if (not (trivial-poly? p))
           #f
           (coeff (first-term (terms p))))))
(define (equ?-poly p q)
  (lambda(p q)
    (cond ((and (trivial-poly? p)
                (trivial-poly? q))
           (equ? (coeff (first-term (terms p)))
                 (coeff (first-term (terms q)))))
          ((same-variable? (variable p) (variable q))
           (let iter ((term-list1 (reverse tl1))
                      (term-list2 (reverse tl2)))
             (cond ((and (empty-termlist? term-list1)
                         (empty-termlist? term-list2))
                    #t)
                   ((or (empty-termlist? term-list1)
                        (empty-termlist? term-list2))
                    #f)
                   ((equ?-terms (first-term term-list1)
                                (first-term term-list2))
                    (iter (rest-terms term-list1)
                          (rest-terms term-list2)))
                   (else #f))))
          (else (error "different vars not supported yet : EQU?-POLY" p
          ↪  q)))))
(put 'equ? '(polynomial polynomial)
     equ?-poly)
(define (make-variable-priority-list poly)
  (cons (variable poly)
        (let rec ((term-list (terms poly)))
          (if (empty-termlist? term-list)
              '()
              (let* ((t (first-term term-list))
                     (c (coeff t))
                     (o (order t)))
                (cond ((is-poly? c)
                       (cons (variable (contents c))
                             (append (rec (rest-terms term-list))
                                     (rec (terms (contents c))))))
                      (else (rec (rest-terms term-list)))))))))
(define (merge-priority-lists l1 l2)
```

```scheme
  (if (null? l2)
      l1
      (let rec ((to-check l1)
                (checking (car l2)))
        (cond ((null? to-check)
               (merge-priority-lists (append l1 (list checking))
                                     (cdr l2)))
              ((same-variable? checking (car to-check))
               (merge-priority-lists l1
                                     (cdr l2)))
              (else (rec (cdr to-check) checking))))))
(define (poly-single-var? wantedvar poly)
  (and (same-variable? (variable poly) wantedvar)
       (let rec ((term-list (terms poly)))
         (if (empty-termlist? term-list)
             #t
             (let ((c (coeff (first-term term-list))))
               (if (not (is-poly? c))
                   (rec (rest-terms term-list))
                   (poly-single-var? wantedvar c)))))))
;;; Example of an inverted poly:
;;; (poly x (order 4, coeff (poly y (5 6)))
;;;         (order 3, coeff (poly y (1 2)))
;;;         (order 0, coeff 7))
;;; (poly y (order 5, coeff (poly x (4 6)))
;;;         (order 1, coeff (poly x (3 2)))
;;;         (order 0, coeff 7))
(define (invert-terms parent-var parent-order term-list)
  (map-terms (lambda(child-order child-coeff)
               (make-sparse-term child-order
                                 (make-polynomial parent-var
                                                  (list
                                                  ↪   (make-sparse-term
                                                  ↪   parent-order
```

```
child-coeff)))))
              term-list))
(define (pad-term-if-needed term parent-priority-list)
  ;; term with tagged poly, priority list -> term
  ;; parent-priority-list starts with the variable this term's order is
  ↪  in
  ;; term: term with tagged poly inside

  ;; pad term so that cross-variable functions will have equal structures
  ;; to thoughtlessly work from. I.e. if your list is '(a b c d e) and
  ;; your term is (5 (poly d (2 1))), we'll return
  ;; '(5 (poly b (poly c (poly d (2 1))))) (nested polys representing
  ↪  dense terms)
  (let* ((child-var (variable (contents (coeff term))))
         (pri (member parent-var parent-priority-list))
         (difference (mem-index child-var pri)))
    (if (= difference 1)
        term
        (let* ((rest-priorities (cdr pri))
               (next-var (car rest-priorities)))
          (make-sparse-term (order term)
                            (make-polynomial next-var
                                             (list (pad-term-if-needed
                                             ↪  (make-sparse-term 0
                                             ↪  (coeff term))
```

```
rest-priorities ))))))))
(define (pad-poly-if-needed poly parent-priority-list)
  ;; untagged poly -> untagged poly
  (let* ((child-var (variable poly))
         (difference (mem-index child-var parent-priority-list)))
    (if (> difference 0)
        (make-poly (car parent-priority-list)
                   (pad-term-if-needed (make-sparse-term 0 (tag poly))
                                       parent-priority-list)))))
(define (reorganize-poly-if-needed poly parent-priority-list)
  ;; untagged poly -> untagged poly
  (let* ((additional-priorities (make-variable-priority-list poly))
         (priorities (merge-priority-lists parent-priority-list
                                           additional-priorities)))
    (let rec ((var (variable poly))
              (remaining-terms (terms poly))
              (done-terms '()))
      (if (empty-termlist? remaining-terms)
          (make-poly var done-terms)
          (let* ((t (first-term remaining-terms))
                 (c (coeff t))
                 (o (order t))
                 (lower-vars (member var priorities)))
            (if (is-poly? c)
                ;; if term is poly, recurse and reorder its terms first
```

```
(let* ((child (rec (variable (contents c))
                   (terms (contents c))
                   '()))
       (child-var (variable child))
       (child-terms (terms child)))
  (if (member child-var lower-vars) ;; make sure poly is lower priority
      ;; if member found, this poly's variable order is not the problem
      (rec var
           (rest-terms remaining-terms)
           (adjoin-term (make-sparse-term o
                                          (tag child))
                        done-terms))
      ;; if member not found, this ordering needs to be inversed
      ;; delay inversion until rest of terms are processed as well
      (let* ((rest (rec var
                        (rest-terms remaining-terms)
                        (adjoin-term (make-sparse-term o
                                                       (tag child))
                                     done-terms)))
             (all-terms (merge-terms done-terms
                                     (adjoin-term t (terms rest))))
             (inverted-terms (invert-terms var o
                                           all-terms)))
        (rec child-var
             (the-empty-termlist)
             (merge-terms child-terms inverted-terms)))))
;; term is not poly, we have nothing to do here
(rec var
```

```
                        (rest-terms remaining-terms)
                        (adjoin-term t done-terms))))))))
                                        ;(define (reorganize-poly-if-needed poly parent-priority-list)
                                        ;  ;; untagged poly -> untagged poly
                                        ;  (let* ((unsorted-priorities (make-variable-priority-list poly))
                                        ;         (priorities (merge-priority-lists parent-priorities
                                        ;                                           unsorted-priorities)))
                                        ;    (let* poly-rec ((p poly))
                                        ;      (let* ((top-var (variable p))
                                        ;             (rest-of-vars (member top-var priorities)))
                                        ;        (receive (are-polys not-polys)
                                        ;          (partition (lambda(t)
                                        ;                       (is-poly? (coeff t)))
                                        ;                     (terms p))
                                        ;          (if (= (length 0) are-polys)
                                        ;              p
                                        ;              (let ((swapped-polys (map (lambda(t)
                                        ;                          (make-sparse-term (order t)
                                        ;                                  (poly-rec (contents (coeff t)))))
                                        ;                       are-polys)))
                                        ;                 )))))
;; table-based reorganization
                                        ;(define (extract-poly-to-table p)
                                        ;  (define ht (make-hash-table 16))
```

```
;  (define (append-one-to-item key to-append)
;    (let ((old (hash-ref ht key '())))
;      (hash-set! ht key (merge-terms old (list to-append)))))
;  (let iter ((tl (terms p)))
;    (if (empty-termlist? tl)
;        ht
;        (let ((t (first-term tl))
;              (top-var (variable p)))
;          (cond ((=zero? (coeff t))
;                 (iter (rest-terms tl)))
;                ((= 0 (order t))
;                 (if (is-poly? (coeff t))
;                     (let ((nested-results (extract-poly-to-table (contents (coeff t)))))
;          (hash-for-each (lambda(key value)
;                (append-one-to-item key value))
;              nested-results)
;                       (iter (rest-terms tl)))
;                     (begin
;  (append-one-to-item 'ZEROTH
;             (make-sparse-term (order t) (coeff t)))
;                       (iter (rest-terms tl)))))
;          ;((is-poly? (coeff t))
```

```
;          ;  (reorganize-poly-if-needed  (coeff t)
;          ;          (order t)
;          ;          top-var
;          ;          parent-priorities))
;               (else (begin  (append-one-to-item top-var  t)
;  (iter (rest-terms tl))))))))
;(define  (invert-poly-table-strict pt)
;  ;; poly table -> poly
;  ;; takes poly (var A) with  terms /all/ coeffs of polys  of (variable B)
;  ;; returns poly (var B) with  terms of (var A)
;  (let ((top-var (variable pt))
;        (term-list (terms pt)))
;    (let ((invalid-terms  (filter (lambda(t)
;     (not (and (is-poly?  (coeff t))
;               (not (eqv?  top-var
;  (variable (coeff t))))))
;     term-list))))
;     (if invalid-terms ;;  TODO: remove when rest of  system is in place
```

```
                                          ;          (error "Poly must contain only terms that are coeffs of a different variable : INVERT-POLY" top-var invalid-terms)
                                          ;          'all-good))
                                          ;    (append-map (lambda(t)(t))
                                          ;                term-list)))
                                          ;(define (reorganize-poly-table-if-needed pt parent-order parent-var parent-priorities)
                                          ;  (let ((priorities (merge-priority-lists parent-priorities
                                          ;                                          (make-variable-priority-list p))))
                                          ;    (map (lambda(p)
                                          ;           (map-terms (lambda(o c)
                                          ;                           );;TODO:
                                          ;                         (cadr p)))
                                          ;         pt)))
(define (add-poly p1 p2)
  (define triv
    ;; if a poly is trivial, return the opposite variable
    (cond ((trivial-poly? p1)
           (variable p2))
          ((trivial-poly? p2)
           (variable p1))
          (else #f)))
  (cond ((or (same-variable? (variable p1) (variable p2))
             triv)
         (make-poly (if triv
                        triv
                        (variable p1))
                    (add-terms (terms p1) (terms p2))))
        (else
         (let ((p2r ((reorganize-poly-if-needed p2
                                                (make-variable-priority-list p1)))))
           (make-poly (variable p1)
                      (add-terms (terms p1) (terms p2r)))))))
(define (mul-poly p1 p2)
  (define triv
    ;; if a poly is trivial, return the opposite variable
```

```
    (cond ((trivial-poly? p1)
           (variable p2))
          ((trivial-poly? p2)
           (variable p1))
          (else #f)))
  (cond ((or (same-variable? (variable p1) (variable p2))
             triv)
         (make-poly (if triv
                        triv
                        (variable p1))
                    (mul-terms (terms p1) (terms p2))))
        (else (error "Polys not in same var: MUL-POLY" (list p1 p2)))))

;; 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 (poly-=zero? p)
  (terms-=zero? (terms p)))
(put '=zero? '(polynomial) poly-=zero?)
(define (neg-poly p)
  (make-poly (variable p)
             (neg-terms (terms p))))
(put 'neg '(polynomial)
     (lambda (p1) (tag (neg-poly p1))))
(define (sub-poly p1 p2)
```

```
  (add-poly p1 (neg-poly p2)))
(put 'sub '(polynomial polynomial)
     (lambda (p1 p2) (tag (sub-poly p1 p2))))
(define (div-poly p1 p2)
  ;; poly, poly -> list of 2 polys
  ;; returns quotient poly and remainder poly
  (if (same-variable? (variable p1) (variable p2))
      (let* ((result (div-terms (terms p1) (terms p2)))
             (quotient (car result))
             (remainder (cadr result)))
        (list (make-poly (variable p1) quotient)
              (make-poly (variable p1) remainder)))
      (error "Polys not in same var: DIV-POLY" (list p1 p2))))
(put 'div '(polynomial polynomial)
     (lambda (p1 p2)
       (let* ((result (div-poly p1 p2))
              (quotient (car result))
              (remainder (cadr result)))
         (list (tag quotient)
               (tag remainder)))))
```

```
;  'done)
(define (make-polynomial var terms)
  ((get 'make 'polynomial) var terms))
```

```
(load "../mattcheck2.scm")
<<op-table>>
<<type-table>>
<<sqrt>>
<<echo>>
<<smarter-type-tagging>>
<<attach-tag-fixed>>
<<list-indices>>
<<tower-predicates>>
<<apply-generic-drop>>
<<generic-arithmetic-complex-parts>>
<<real-package>>
<<eqzero?>>
<<real-package>>
<<tower-coercions-polynomial>>

<<mem-index>>
<<polynomial-mixed-variables-top>>
<<polynomial-mixed-variables-bottom>>

<<generic-negation>>

```

```
(install-rational-package)
(install-rectangular-package)
(install-polar-package)
(install-complex-package)
(install-scheme-number-package)
(install-real-package)
(install-equ?)
(install-=zero?)
(install-polynomial-package)
(install-negation)

<<gen-math-tests>>

(mattcheck "raise success"
           (raise 78)
           (make-rational 78 1))
(mattcheck "raise fail"
           (raise (make-polynomial 'x '(3 2)))
           #f)
(mattcheck "apply-generic 1"
           (add 1 (make-rational 3 5))
           (make-rational 8 5))
(mattcheck "apply-generic 2"
           (add (make-real 2)
                (make-complex-from-mag-ang 3 5))
           (make-complex-from-real-imag 2.850986556389679
           ↪  -2.8767728239894153))
(mattcheck "drop 1"
           (drop (make-complex-from-real-imag 5 0))
           5)
(mattcheck "drop 2"
           (drop (make-real 5))
           5)
(mattcheck "drop 3"
           (drop (make-rational 5 1))
           5)
(mattcheck "complex with nested rational"
           (make-complex-from-real-imag 2 (make-rational 3 5))
           (cons 'complex (cons 'rectangular (cons 2 (cons 'rational
           ↪  (cons 3 5))))))
(mattcheck "math on complex with nested rational"
           (add (make-complex-from-real-imag 2 (make-rational 3 5))
                (make-complex-from-mag-ang (make-rational 3 5) 2))
           (cons 'complex (cons 'rectangular
                                (cons (cons 'rational (cons
                                ↪  8.751559490358574 5))
                                      (cons 'rational (cons
                                      ↪  28.639461402385223 25))))))
```

```
(mattcheck ”polynomial = zero”
           (=zero? (make-polynomial 'x '((0 0)))))
(mattcheck ”polynomial != zero” #f
           (=zero? (make-polynomial 'x '((1 1)(0 1)))))

(define p1 '((100 1)(2 2)(0 1)))
(define p2 (list (list 7 (make-polynomial 'x '((3 2))))))
(define p3 '((100 1)(2 1)(0 3)))
(define p4 '((100 -1)(2 -2)(0 -1)))
(define p5 '(8 2 5 -3 -7))
(define p7 '((1 1)(0 1)))
(define p6 '((3 1)(0 1)))
(define p66 '(1 0 0 1))
(define p77 '(1 1))
(define p8 (make-poly 'a
                           (list (list 9 (make-polynomial 'b p3))
                                 (list 8 (make-polynomial 'c p5))
                                 (list 1 (make-polynomial 'd (list (list 0 (make-polynomial 'f '(3 2))))))
                                 (list 0 (make-polynomial 'e (list 2 0))))))
  (mattcheck ”polynomial addition”
             (add (make-polynomial 'x p1)
                  (make-polynomial 'x p1))
             '(polynomial x (100 2)(2 4)(0 2)))
  (mattcheck ”polynomial multiplication”
             (mul (make-polynomial 'x p1)
                  (make-polynomial 'x p1))
             '(polynomial x (200 1) (102 4) (100 2) (4 4) (2 4) (0 1)))
  (mattcheck ”nested polynomial addition”
             (add (make-polynomial 'x p2)
                  (make-polynomial 'x p1))
             '(polynomial x (100 1) (7 (polynomial x (3 2))) (2 2) (0 1)))
  (mattcheck ”nested polynomial mult”
             (mul (make-polynomial 'x p2)
                  (make-polynomial 'x p1))
             '(polynomial x (107 (polynomial x (3 2))) (9 (polynomial x (3 4))) (7 (polynomial x (3 2)))))
  (mattcheck ”negation scheme-number”
             (neg 5) -5)
  (mattcheck ”negation rational”
             (neg (make-rational 1 5))
             (make-rational -1 5))
  (mattcheck ”negation rectangular complex”
             (neg (make-complex-from-real-imag 3 7))
             (make-complex-from-real-imag -3 -7))
  (mattcheck ”negation polar complex”
```

```
                   (neg (make-complex-from-mag-ang 3 7))
                   (make-complex-from-mag-ang -3 7))

    (mattcheck "negation polynomials"
                   (neg (make-polynomial 'x p1))
                   (make-polynomial 'x p4))
    (mattcheck "polynomial subtraction"
                   (sub (make-polynomial 'x p1)
                        (make-polynomial 'x p3))
                   '(polynomial x 1 0 -2))
    (mattcheck "polynomial division 1"
                   (div (make-polynomial 'x p6)
                        (make-polynomial 'x p7))
                   '((polynomial x 1 -1 1) (polynomial x)))
    (mattcheck "polynomial division 2"
                   (div (make-polynomial 'x p7)
                        (make-polynomial 'x p6))
                   '((polynomial x) (polynomial x 1 1)))
    (mattcheck "polynomial division 3"
                   (div (make-polynomial 'x p66)
                        (make-polynomial 'x p77))
                   '((polynomial x 1 -1 1) (polynomial x)))

    (mattcheck "poly projection good"
                   (project (make-polynomial 'useless '((0 7))))
                   7)
    (mattcheck "poly projection bad"
                   (project (make-polynomial 'useless '((1 7))))
                   #f)
    (mattcheck "raise to poly"
                   (raise (make-complex-from-mag-ang 78 64))
                   '(polynomial DISCARD-ME (complex polar 78 . 64)))
    (mattcheck "polynomials in multiple variables 1"
                   (add (make-polynomial 'x p1)
                        (make-polynomial 'y p1))
                   'unknown)
```