

# A Journey Through SICP

Notes, exercises and analyses of Abelson and Sussman

ProducerMatt

November 26, 2022

# Contents

<b>1</b>	<b>Introduction Notes</b>	<b>6</b>		
1.1	Text Foreword . . . . .	6		
1.2	Preface, 1e . . . . .	6		
<b>2</b>	<b>Chapter 1: Building Abstractions with Procedures</b>	<b>7</b>		
2.1	1.1: The Elements of Programming . .	7		
2.2	1.1.1: Expressions . . . . .	7		
2.3	1.1.3: Evaluating Combinations . . . .	8		
2.4	1.1.4: Compound Procedures . . . . .	8		
2.5	1.1.5: The Substitution Model for Procedure Application . . . . .	8		
2.6	1.1.6: Conditional Expressions and Predicates . . . . .	9		
2.7	Exercise 1.1 . . . . .	9		
	2.7.1 Question . . . . .	9		
	2.7.2 Answer . . . . .	10		
2.8	Exercise 1.2 . . . . .	10		
	2.8.1 Question . . . . .	10		
	2.8.2 Answer . . . . .	10		
2.9	Exercise 1.3 . . . . .	10		
	2.9.1 Question . . . . .	10		
	2.9.2 Answer . . . . .	10		
2.10	Exercise 1.4 . . . . .	10		
	2.10.1 Question . . . . .	10		
	2.10.2 Answer . . . . .	10		
2.11	Exercise 1.5 . . . . .	11		
	2.11.1 Question . . . . .	11		
	2.11.2 Answer . . . . .	11		
2.12	1.1.7: Example: Square Roots by Newton's Method . . . . .	11		
2.13	1.1.8: Procedures as Black-Box Abstractions . . . . .	11		
2.14	Exercise 1.6 . . . . .	11		
	2.14.1 Text code . . . . .	11		
	2.14.2 Question . . . . .	12		
	2.14.3 Answer . . . . .	12		
2.15	Exercise 1.7 . . . . .	12		
	2.15.1 Text . . . . .	12		
	2.15.2 Question . . . . .	12		
	2.15.3 Diary . . . . .	13		
	2.15.4 Answer . . . . .	14		
2.16	Exercise 1.8 . . . . .	14		
	2.16.1 Question . . . . .	14		
	2.16.2 Diary . . . . .	14		
	2.16.3 Answer . . . . .	15		
2.17	1.2: Procedures and the Processes They Generate . . . . .	15		
2.18	1.2.1: Linear Recursion and Iteration .	15		
2.19	Exercise 1.9 . . . . .	16		
	2.19.1 Question . . . . .	16		
	2.19.2 Answer . . . . .	16		
2.20	Exercise 1.10 . . . . .	16		
	2.20.1 Question . . . . .	16		
	2.20.2 Answer . . . . .	17		
2.21	1.2.2: Tree Recursion . . . . .	17		
	2.21.1 Example: Counting change . .	18		
2.22	Exercise 1.11 . . . . .	18		
	2.22.1 Question . . . . .	18		
	2.22.2 Answer . . . . .	18		
2.23	Exercise 1.12 . . . . .	19		
	2.23.1 Question . . . . .	19		
	2.23.2 Answer . . . . .	19		
2.24	Exercise 1.13 . . . . .	20		
	2.24.1 Question . . . . .	20		
	2.24.2 Answer . . . . .	20		
2.25	1.2.3: Orders of Growth . . . . .	20		
2.26	Exercise 1.14 . . . . .	21		
	2.26.1 Text . . . . .	21		
	2.26.2 Question A . . . . .	21		
	2.26.3 Answer . . . . .	21		
	2.26.4 Question B . . . . .	22		
	2.26.5 Answer B . . . . .	22		
2.27	Exercise 1.15 . . . . .	24		
	2.27.1 Question A . . . . .	24		
	2.27.2 Answer A . . . . .	24		
	2.27.3 Question B . . . . .	24		
	2.27.4 Answer B . . . . .	24		
2.28	Exercise 1.16 . . . . .	26		
	2.28.1 Text . . . . .	26		
	2.28.2 Question . . . . .	26		
	2.28.3 Diary . . . . .	26		
	2.28.4 Answer . . . . .	27		
2.29	Exercise 1.17 . . . . .	27		
	2.29.1 Question . . . . .	27		
	2.29.2 Answer . . . . .	28		
2.30	Exercise 1.18 . . . . .	28		
	2.30.1 Question . . . . .	28		
	2.30.2 Diary . . . . .	28		
	2.30.3 Answer . . . . .	28		

2.31	Exercise 1.19	29	2.46.1	Question	46
2.31.1	Question	29	2.46.2	Answer	46
2.31.2	Diary	29	2.47	Exercise 1.31	46
2.31.3	Answer	30	2.47.1	Question A.1	46
2.32	1.2.5: Greatest Common Divisor	30	2.47.2	Answer A.1	46
2.33	Exercise 1.20	31	2.47.3	Question A.2	46
2.33.1	Text	31	2.47.4	Answer A.2	46
2.33.2	Question	31	2.47.5	Question A.3	47
2.33.3	Answer	31	2.47.6	Answer A.3	47
2.34	1.2.6: Example: Testing for Primality	32	2.47.7	Question B	47
2.35	Exercise 1.21	33	2.47.8	Answer B	47
2.35.1	Text	33	2.48	Exercise 1.32	47
2.35.2	Question	33	2.48.1	Question A	47
2.35.3	Answer	33	2.48.2	Answer A	48
2.36	Exercise 1.22	33	2.48.3	Question B	48
2.36.1	Question	33	2.48.4	Answer B	48
2.36.2	Answer	34	2.49	Exercise 1.33	48
2.37	Exercise 1.23	36	2.49.1	Question A	48
2.37.1	Question	36	2.49.2	Answer A	48
2.37.2	A Comedy of Error (just the one)	36	2.49.3	Question B	49
2.37.3	Answer	37	2.50	1.3.2: Constructing Procedures Using lambda	49
2.38	Exercise 1.24	39	2.51	Exercise 1.34	49
2.38.1	Text	39	2.51.1	Question	49
2.38.2	Question	39	2.51.2	Answer	50
2.38.3	Answer	39	2.52	1.3.3 Procedures as General Methods	50
2.39	Exercise 1.25	41	2.53	Exercise 1.35	50
2.39.1	Question	41	2.53.1	Text	50
2.39.2	Answer	41	2.53.2	Question	50
2.40	Exercise 1.26	41	2.53.3	Answer	50
2.40.1	Question	41	2.54	Exercise 1.36	51
2.40.2	Answer	42	2.54.1	Question	51
2.41	Exercise 1.27	42	2.54.2	Answer	51
2.41.1	Question	42	2.55	Exercise 1.37	51
2.41.2	Answer	42	2.55.1	Question A	51
2.42	Exercise 1.28	42	2.55.2	Answer A	51
2.42.1	Question	42	2.55.3	Question B	52
2.42.2	Analysis	43	2.55.4	Answer B	52
2.42.3	Answer	43	2.55.5	Question C	52
2.43	1.3: Formulating Abstractions with Higher-Order Procedures	45	2.55.6	Answer C	52
2.44	1.3.1: Procedures as Arguments	45	2.56	Exercise 1.38	52
2.45	Exercise 1.29	45	2.56.1	Question	52
2.45.1	Text	45	2.56.2	Answer	53
2.45.2	Question	45	2.57	Exercise 1.39	53
2.45.3	Answer	45	2.57.1	Question	53
2.46	Exercise 1.30	46	2.57.2	Answer	53

2.58	1.3.4 Procedures as Returned Values . . .	53	3.8	Exercise 2.6 . . . . .	66
2.59	Exercise 1.40 . . . . .	53	3.8.1	Question . . . . .	66
	2.59.1 Text . . . . .	53	3.8.2	Answer . . . . .	66
	2.59.2 Question . . . . .	54	3.9	Exercise 2.7 . . . . .	67
	2.59.3 Answer . . . . .	54	3.9.1	Text . . . . .	67
2.60	Exercise 1.41 . . . . .	54	3.9.2	Question . . . . .	67
	2.60.1 Question . . . . .	54	3.9.3	Answer . . . . .	67
	2.60.2 Answer . . . . .	54	3.10	Exercise 2.8 . . . . .	67
2.61	Exercise 1.42 . . . . .	54	3.10.1	Question . . . . .	67
	2.61.1 Question . . . . .	54	3.10.2	Answer . . . . .	68
	2.61.2 Answer . . . . .	54	3.11	Exercise 2.9 . . . . .	68
2.62	Exercise 1.43 . . . . .	55	3.11.1	Question . . . . .	68
	2.62.1 Question . . . . .	55	3.11.2	Answer . . . . .	68
	2.62.2 Answer . . . . .	55	3.12	Exercise 2.10 . . . . .	69
2.63	Exercise 1.44 . . . . .	55	3.12.1	Question . . . . .	69
	2.63.1 Question . . . . .	55	3.12.2	Answer . . . . .	69
	2.63.2 Answer . . . . .	55	3.13	Exercise 2.11 . . . . .	69
2.64	Exercise 1.45 . . . . .	55	3.13.1	Question . . . . .	69
	2.64.1 Question . . . . .	55	3.13.2	Answer . . . . .	70
	2.64.2 Answer . . . . .	56	3.14	Exercise 2.12 . . . . .	71
2.65	Exercise 1.46 . . . . .	57	3.14.1	Question . . . . .	71
	2.65.1 Question . . . . .	57	3.14.2	Answer . . . . .	72
	2.65.2 Answer . . . . .	57	3.15	Exercise 2.13 . . . . .	72
			3.15.1	Question . . . . .	72
			3.15.2	Answer . . . . .	73
<b>3 Chapter 2: Building Abstractions with Data</b>	<b>58</b>		3.16	Exercise 2.14 . . . . .	73
3.1	2.1.1: Example: Arithmetic Operations for Rational Numbers . . . . .	58	3.16.1	Question . . . . .	73
3.2	Exercise 2.1 . . . . .	58	3.16.2	Answer . . . . .	74
	3.2.1 Text . . . . .	58	3.17	Exercise 2.15 . . . . .	74
	3.2.2 Question . . . . .	59	3.17.1	Question . . . . .	74
	3.2.3 Answer . . . . .	59	3.17.2	Answer . . . . .	74
3.3	Exercise 2.2 . . . . .	59	3.18	Exercise 2.16 . . . . .	75
	3.3.1 Question . . . . .	59	3.18.1	Question . . . . .	75
	3.3.2 Answer . . . . .	60	3.18.2	Answer . . . . .	75
3.4	Exercise 2.3 . . . . .	60	3.19	2.2: Hierarchical Data and the Closure Property . . . . .	75
	3.4.1 Question . . . . .	60	3.20	2.2.1: Representing Sequences . . . . .	75
	3.4.2 Answer 1 . . . . .	61	3.21	Exercise 2.17 . . . . .	76
	3.4.3 Answer 2 . . . . .	62	3.21.1	Question . . . . .	76
3.5	2.1.3: What Is Meant by Data? . . . . .	64	3.21.2	Answer . . . . .	76
3.6	Exercise 2.4 . . . . .	64	3.22	Exercise 2.18 . . . . .	76
	3.6.1 Question . . . . .	64	3.22.1	Question . . . . .	76
	3.6.2 Answer . . . . .	64	3.22.2	Answer . . . . .	76
3.7	Exercise 2.5 . . . . .	64	3.23	Exercise 2.19 . . . . .	76
	3.7.1 Question . . . . .	64	3.23.1	Question . . . . .	76
	3.7.2 Answer . . . . .	65	3.23.2	Answer . . . . .	77

3.24	Exercise 2.20 . . . . .	77	3.36.2	Answer . . . . .	87
	3.24.1 Question . . . . .	77	3.37	2.2.3: Sequences as Conventional In-	
	3.24.2 Answer . . . . .	77		terfaces . . . . .	88
3.25	Exercise 2.21 . . . . .	78	3.38	Exercise 2.33: The flexibility of	
	3.25.1 Question . . . . .	78		(accumulate) . . . . .	88
	3.25.2 Answer . . . . .	78	3.38.1	Text Definitions . . . . .	88
3.26	Exercise 2.22 . . . . .	79	3.38.2	Question . . . . .	88
	3.26.1 Questions . . . . .	79	3.38.3	Answer . . . . .	88
	3.26.2 Answer . . . . .	79	3.39	Exercise 2.34 . . . . .	89
3.27	Exercise 2.23 . . . . .	79	3.39.1	Question . . . . .	89
	3.27.1 Question . . . . .	79	3.39.2	Answer . . . . .	89
	3.27.2 Answer . . . . .	80	3.40	Exercise 2.35 . . . . .	90
3.28	Exercise 2.24 . . . . .	80	3.40.1	Question . . . . .	90
	3.28.1 Text Definitions . . . . .	80	3.40.2	Answer . . . . .	90
	3.28.2 Question . . . . .	80	3.41	Exercise 2.36: Accumulate across mul-	
	3.28.3 Answer . . . . .	80		tiplists . . . . .	90
3.29	Exercise 2.25 . . . . .	81	3.41.1	Question . . . . .	90
	3.29.1 Question . . . . .	81	3.41.2	Answers . . . . .	90
3.30	Exercise 2.26 . . . . .	81	3.42	Exercise 2.37: Enter the matrices . . .	90
	3.30.1 Question . . . . .	81	3.42.1	Question . . . . .	90
	3.30.2 Answer . . . . .	81	3.42.2	Answer . . . . .	91
3.31	Exercise 2.27 . . . . .	81	3.43	Exercise 2.38: fold-right . . . . .	91
	3.31.1 Question . . . . .	81	3.43.1	Question A . . . . .	91
	3.31.2 Answer . . . . .	82	3.43.2	Answer A . . . . .	92
3.32	Exercise 2.28 . . . . .	82	3.43.3	Question B . . . . .	92
	3.32.1 Question . . . . .	82	3.43.4	Answer B . . . . .	92
	3.32.2 Answer . . . . .	82	3.44	Exercise 2.39: reverse via fold . . . . .	92
3.33	Exercise 2.29: Binary Mobiles . . . . .	82	3.44.1	Question . . . . .	92
	3.33.1 Text Definitions . . . . .	82	3.44.2	Answer . . . . .	92
	3.33.2 Question A: Selectors . . . . .	82	3.45	Exercise 2.40: unique-pairs . . . . .	93
	3.33.3 Answer A . . . . .	82	3.45.1	Text Definitions . . . . .	93
	3.33.4 Question B: total-weight . . . . .	82	3.45.2	Question . . . . .	93
	3.33.5 Answer B . . . . .	83	3.45.3	Answer . . . . .	93
	3.33.6 Question C: Balancing . . . . .	83	3.46	Exercise 2.41: Ordered triples of posi-	
	3.33.7 Answer C . . . . .	83		tive integers . . . . .	94
	3.33.8 Question D: Implementation		3.46.1	Question . . . . .	94
	shakeup . . . . .	84	3.46.2	Answer . . . . .	94
	3.33.9 Answer D . . . . .	85	3.47	Exercise 2.42: Eight Queens . . . . .	94
3.34	Exercise 2.30 . . . . .	85	3.47.1	Question . . . . .	94
	3.34.1 Question . . . . .	85	3.47.2	Answer . . . . .	95
	3.34.2 Answer . . . . .	86	3.48	Exercise 2.43: Louis' queens . . . . .	95
3.35	Exercise 2.31 . . . . .	86	3.48.1	Question . . . . .	95
	3.35.1 Question . . . . .	86	3.48.2	Answer . . . . .	96
	3.35.2 Answer . . . . .	86	3.49	2.2.4: Example: A Picture Language .	96
3.36	Exercise 2.32 . . . . .	87	3.50	Exercise 2.44: up-split . . . . .	98
	3.36.1 Question . . . . .	87	3.50.1	Text Definitions . . . . .	98

3.50.2	Question	98
3.50.3	Answer	98
3.51	Exercise 2.45: Generalized splitting	98
3.51.1	Question	98
3.51.2	Answer	99
3.52	Exercise 2.46: Defining vectors	99
3.52.1	Question	99
3.52.2	Answer	99
3.53	Exercise 2.47: Defining frames	100
3.53.1	Question	100
3.53.2	Answer	100
3.54	Exercise 2.48: Line segments	100
3.54.1	Question	100
3.54.2	Answer	100
3.55	Exercise 2.49: Primitive painters	100
3.55.1	Text Definitions	100
3.55.2	Question	101
3.55.3	Answer	101
3.56	Exercise 2.50: Transforming painters	102
3.56.1	Text Definitions	102
3.56.2	Question	103
3.56.3	Answer	104
3.57	Exercise 2.51	104
3.57.1	Question	104
3.57.2	Answer	105
3.58	2.2.4 continued	106
3.59	Exercise 2.52	106
3.59.1	Question A	106
3.59.2	Answer	106

# 1 Introduction Notes

## 1.1 Text Foreword

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

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

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

## 1.2 Preface, 1e

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

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

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

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

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

**Imperative knowledge** *How to follow a process*.

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

### 1. Techniques for controlling complexity

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

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

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

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

Black-box abstraction

- (a) Start with primitive objects of procedures and data.

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

**Conventional interfaces** Agreed-upon ways of connecting things together.

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

#### Object-oriented programming

thinking of your structure as a society of discrete but interacting parts.

#### Operations on aggregates

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

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

## 2 Chapter 1: Building Abstractions with Procedures

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

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

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

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

### 2.1 1.1: The Elements of Programming

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

**primitive expressions** the simplest entities the language is concerned with

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

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

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

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

```
1 (define (square x)
2   (* x x))
```

```
1 <<square>>
2 (square 5)
```

x is the formal parameter and 5 is the argument.

### 2.2 1.1.1: Expressions

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

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

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

```

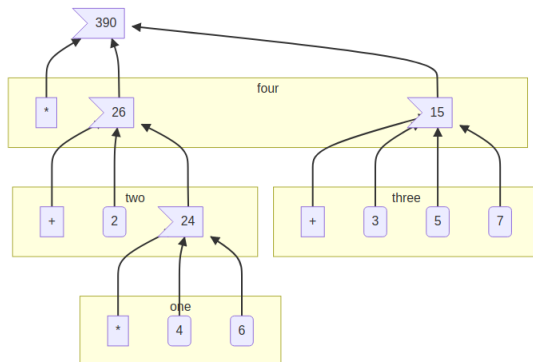
1 (define (square x) (* x x)) ; to square
  ↪ something, multiply it by itself
2 ; now it can be applied or used in other
  ↪ definitions:
3 (square 4) ; => 16
4
5 (define (sum-of-squares x y)
6   (+ (square x) (square y)))
7 (sum-of-squares 3 4) ; => 25

```

## 2.3 1.1.3: Evaluating Combinations

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

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



## 2.4 1.1.4: Compound Procedures

We have identified the following in Lisp:

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

Procedure definitions give a name to a compound procedure.

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

## 2.5 1.1.5: The Substitution Model for Procedure Application

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

```

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

```

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

Evaluations can be done in different orders.

**Applicative order** evaluates the operator and operands, and then applies the

resulting procedure to the resulting arguments. In other words, reducing, then expanding, then reducing.

**Normal order** substitutes expressions until it obtains an expression involving

only primitive operators, or until it can't substitute any further, and then evaluates. This results in expanding the expression completely before doing any



reduction, which results in some repeated evaluations.

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

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

```

1 ; Applicative evaluation
2 (f 5)
3 (sum-of-squares (+ a 1) (* a 2))
4 (sum-of-squares (+ 5 1) (* 5 2))
5 (sum-of-squares 6 10)
6 (+ (square x)(square y))
7 (+ (square 6)(square 10))
8 (+ (* 6 6)(* 10 10))
9 (+ 36 100)
10 136
11 ; Normal evaluation
12 (f 5)
13 (sum-of-squares (+ a 1) (* a 2))
14 (sum-of-squares (+ 5 1) (* 5 2))
15 (+ (square (+ 5 1)) (square (* 5 2)))
16 (+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2)))
17 (+ (* 6 6) (* 10 10))
18 (+ 36 100)
19 136

```

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

## 2.6 1.1.6: Conditional Expressions and Predicates

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

```

1 (cond
2   (predicate1 consequent1)
3   ...)

```

```
(predicateN consequentN))
```

A shorter form of conditional:

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

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

Combining predicates:

```

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

```

A small clarification:

```

1 (define A (* 5 5))
2 (define (D) (* 5 5))
3 A ; => 25
4 D ; => compound procedure D
5 (D) ; => 25 (result of executing procedure D)

```

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

## 2.7 Exercise 1.1

### 2.7.1 Question

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

## 2.7.2 Answer

```

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

```

## 2.8 Exercise 1.2

### 2.8.1 Question

Translate the following expression into prefix form:

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

### 2.8.2 Answer

```

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

```

1/75

## 2.9 Exercise 1.3

### 2.9.1 Question

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

## 2.9.2 Answer

```

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

```

```

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

```

```

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

```

## 2.10 Exercise 1.4

### 2.10.1 Question

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

```

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

```

### 2.10.2 Answer

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

```

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

```

## 2.11 Exercise 1.5

### 2.11.1 Question

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

```

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

```

Then he evaluates the expression:

```

1  (test 0 (p))

```

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

### 2.11.2 Answer

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

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

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

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

## 2.13 1.1.8: Procedures as Black-Box Abstractions

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

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

Suppressing these details are particularly helpful:

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

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

## 2.14 Exercise 1.6

### 2.14.1 Text code

```

1  (define (abs x)

```

```

2 (if (< x 0)
3     (- x)
4     x))

```

```

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

```

```

1 <<average>>
2 (define (improve guess x)
3   (average guess (/ x guess)))
4
5 <<square>>
6 <<abs>>
7 (define (good-enough? guess x)
8   (< (abs (- (square guess) x)) 0.001))
9
10 (define (sqrt-iter guess x)
11   (if (good-enough? guess x)
12       guess
13       (sqrt-iter (improve guess x) x)))
14
15 (define (sqrt x)
16   (sqrt-iter 1.0 x))

```

### 2.14.2 Question

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

```

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

```

Eva demonstrates the program for Alyssa:

```

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

```

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

```

1 (define (sqrt-iter guess x)
2   (new-if (good-enough? guess x)
3           guess
4           (sqrt-iter (improve guess x) x)))

```

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

### 2.14.3 Answer

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

## 2.15 Exercise 1.7

### 2.15.1 Text

```

1 (define (mean-square x y)
2   (average (square x) (square y)))

```

### 2.15.2 Question

The `good-enough?` test used in computing square roots will not be very effective for finding the square roots of very small numbers. Also, in real computers, arithmetic operations are almost always performed with limited precision. This makes our test inadequate for very large numbers. Explain these statements, with examples showing how the test fails for small and large numbers. An alternative strategy for implementing `good-enough?` is to watch how `guess` changes from one iteration to the next

and to stop when the change is a very small fraction of the guess. Design a square-root procedure that uses this kind of end test. Does this work better for small and large numbers?

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

### 2.15.3 Diary

#### 1. Solving

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

```

1 <<txt-sqrt>>
2 (define (inferior-good-enough? guess
3   ↪ lastguess)
4   (<=
5     (abs (-
6       (/ lastguess guess)
7       1))
8     0.000000000000001)) ; dx
9 (define (new-sqrt-iter guess x lastguess)
10  ↪ ;; Memory of previous value
11  (if (inferior-good-enough? guess
12    ↪ lastguess)
13    guess
14    (new-sqrt-iter (improve guess x) x
15    ↪ guess)))
15 (define (new-sqrt x)
16   (new-sqrt-iter 1.0 x 0))

```

This solution can correctly find small and large numbers:

```

1 <<inferior-good-enough>>
2 (new-sqrt 1000000000000000)

```

3162277.6601683795

```

1 <<try-these>>
2 <<inferior-good-enough>>
3 (try-these new-sqrt '(0.01 0.0001
4   ↪ 0.000001 0.00000001 0.0000000001))

```

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

```

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

```

#### 2. Improving (sqrt) by avoiding extra (improve) call

##### (a) Non-optimized

```

1 (use-modules (ice-9 format))
2 (load "../mattbench.scm")
3 (define (average x y)
4   (/ (+ x y) 2))
5 (define (improve guess x)
6   (average guess (/ x guess)))
7 (define (good-enough? guess x)
8   (= (improve guess x) guess)) ;;
9   ↪ improve call 1
10 (define (sqrt-iter guess x)
11   (if (good-enough? guess x)
12     guess
13     (sqrt-iter (improve guess x)
14     ↪ x))) ;; call 2
14 (define (sqrt x)
15   (sqrt-iter 1.0 x))
16 (newline)
17 (display (mattbench (λ() (sqrt
18   ↪ 69420)) 400000000))
19 (newline)
20 ;; 4731.30 <- Benchmark results

```

##### (b) Optimized

```

1 (use-modules (ice-9 format))
2 (load "../mattbench.scm")
3 (define (average x y)
4   (/ (+ x y) 2))
5 (define (improve guess x)

```

```

6 (average guess (/ x guess)))
7 (define (good-enough? guess
  ↪ nextguess x)
8   (= nextguess guess))
9 (define (sqrt-iter guess x)
10  (let ((nextguess (improve guess
  ↪ x)))
11    (if (good-enough? guess
  ↪ nextguess x)
12        guess
13        (sqrt-iter nextguess x))))
14 (define (sqrt x)
15   (sqrt-iter 1.0 x))
16 (newline)
17 (display (mattbench (λ() (sqrt
  ↪ 69420)) 400000000))
18 (newline)

```

(c) Benchmark results

Unoptimized	4731.30
Optimized	2518.44

```

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

```

```

1 <<try-these>>
2 <<sqrt>>
3 (try-these sqrt '(0.01 0.0001 0.000001
  ↪ 0.00000001 0.0000000001))

```

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

## 2.15.4 Answer

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

```

1 <<txt-sqrt>>
2 <<try-these>>
3 (try-these sqrt 0.01 0.0001 0.000001
  ↪ 0.00000001 0.0000000001)

```

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

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

```

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

```

1000000.0

So, my definition of sqrt:

## 2.16 Exercise 1.8

### 2.16.1 Question

Newton's method for cube roots is based on the fact that if  $y$  is an approximation to the cube root of  $x$ , then a better approximation is given by the value

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

Use this formula to implement a cube-root procedure analogous to the `square-root` procedure. (In 1.3.4 we will see how to implement Newton's method in general as an abstraction of these square-root and cube-root procedures.)

### 2.16.2 Diary

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

```

1 <<square>>
2 <<try-these>>
3 (define (cb-good-enough? guess x)
4   (= (cb-improve guess x) guess))
5 (define (cb-improve guess x)
6   (/
7     (+
8       (/ x (square guess))
9       (* guess 2))
10    3))
11 (define (cbrt-iter guess x counter)
12   (if (or (cb-good-enough? guess x) (> counter
13     ↪ 100))
14       guess
15       (begin
16         (cbrt-iter (cb-improve guess x) x (+ 1
17     ↪ counter)))))
18 (define (cbrt x)
19   (cbrt-iter 1.0 x 0))
20 (try-these cbrt 7 32 56 100)

```

7	1.912931182772389
32	3.174802103936399
56	3.825862365544778
100	4.641588833612779

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

### 2.16.3 Answer

```

1 <<square>>
2 (define (cb-good-enough? nextguess guess
3   ↪ lastguess x)
4   (or (= nextguess guess)
5       (= nextguess lastguess)))
6 (define (cb-improve guess x)
7   (/
8     (+

```

```

8     (/ x (square guess))
9     (* guess 2))
10    3))
11 (define (cbrt-iter guess lastguess x)
12   (define nextguess (cb-improve guess x))
13   (if (cb-good-enough? nextguess guess
14     ↪ lastguess x)
15       nextguess
16       (cbrt-iter nextguess guess x)))
17 (define (cbrt x)
18   (cbrt-iter 1.1 9999 x))

```

```

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

```

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

## 2.17 1.2: Procedures and the Processes They Generate

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

### 2.18 1.2.1: Linear Recursion and Iteration

Consider these two procedures for obtaining factorials:

```

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

```

```

12      (* counter product)
13      (+ counter 1)
14      max-count)))
15
16 (fact-iter 1 1 n))

```

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

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

**recursive procedure** is a procedure whose definition refers to itself.

**recursive process** is a process that evolves recursively.

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

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

## 2.19 Exercise 1.9

### 2.19.1 Question

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

```

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

```

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

### 2.19.2 Answer

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

#### 1. recursive procedure

```

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

```

#### 2. iterative procedure

```

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

```

## 2.20 Exercise 1.10

### 2.20.1 Question

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



```

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

```

What are the values of the following expressions?

```

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

```

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

```

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

```

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

## 2.20.2 Answer

1. f

```

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

```

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

$$f(n) = 2n$$

2. g

```

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

```

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

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

3. h

```

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

```

1	2
2	4
3	16
4	65536

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

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

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

## 2.21 1.2.2: Tree Recursion

Consider a recursive procedure for computing Fibonacci numbers:

```

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

```

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

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

```

1 (define (fib n)
2   (fib-iter 1 0 n))
3 (define (fib-iter a b count)
4   (if (= count 0) b (fib-iter (+ a b) a (-
    ↪ count 1))))

```

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

### 2.21.1 Example: Counting change

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

## 2.22 Exercise 1.11

### 2.22.1 Question

A function  $f$  is defined by the rule that:

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

and

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

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

### 2.22.2 Answer

#### 1. Recursive

```

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

```

```

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

```

1	1
3	4
5	25
10	1892

#### 2. Iterative

##### (a) Attempt 1

```

1 ;; This seems like it could be better
2 (define (fi n)
3   (define (formula l)
4     (let ((a (car l))
5           (b (cadr l))
6           (c (caddr l)))
7       (+ a
8          (* 2 b)
9          (* 3 c))))
10  (define (iter l i)
11    (if (= i n)
12        (car l)
13        (iter (cons (formula l) l)
14               (+ 1 i))))
15  (if (< n 3)
16      n
17      (iter '(2 1 0) 2)))

```

```

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

```

```

      1      1
     3      4
    5     25
   10    1892

```

It works but it seems wasteful.

(b) Attempt 2

```

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

```

```

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

```

```

      1      1
     3      4
    5     25
   10    1892

```

I like that better.

## 2.23 Exercise 1.12

### 2.23.1 Question

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

```

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

```

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

### 2.23.2 Answer

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

```

1 (define (pascal x y)
2   (if (or (= x 0)
3           (= y 0))
4       1
5       (+ (pascal (- x 1) y)
6          (pascal x (- y 1)))))

```

```

1 <<try-these>>
2 <<pascal-rec>>
3 (let ((l (iota 8)))
4   (map (λ (row)
5         (map (λ (xy)
6               (apply pascal xy))
7              row))
8   (map (λ (x)
9         (map (λ (y)
10              (list x y))
11              l))
12   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 optional

### 2.24.1 Question

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

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

### 2.24.2 Answer

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

1. Fibonacci number generator

```

1 (define (fib-iter n)
2   (define (iter i a b)
3     (if (= i n)
4         b
5         (iter (+ i 1)
6               b
7               (+ a b))))
8   (if (<= n 2)
9       1
10      (iter 2 1 1)))

```

2. Various algorithms relating to the question

```

1 <<sqr>>
2 (define sqrt5
3   (sqrt 5))
4 (define phi
5   (/ (+ 1 sqrt5) 2))
6 (define upilon
7   (/ (- 1 sqrt5) 2))
8 (define (fib-phi n)
9   (/ (- (expt phi n)
10         (expt upilon n))
11      sqrt5))

```

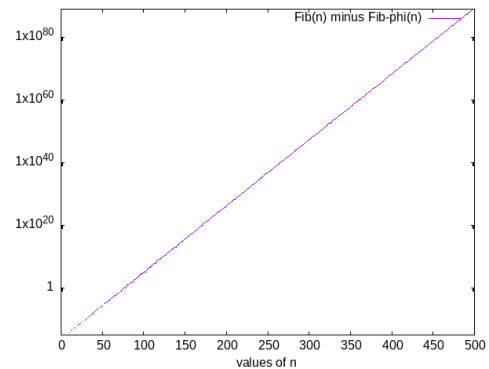
```

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

```

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

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



## 2.25 1.2.3: Orders of Growth

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

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

For example:

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

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

Some algorithms we've already seen:

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

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

**Tree-recursive** means in general, time is proportional to the number of nodes, space is proportional to the depth of the tree. In the Fibonacci algorithm example,  $\Theta(n)$  and time  $\Theta(\Upsilon^n)$  where  $\Upsilon$  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

### 2.26.1 Text

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

```
1 (define (count-change amount)
2   (cc amount 5))
3
4 (define (cc amount kinds-of-coins)
5   (cond ((= amount 0) 1)
6         ((or (< amount 0)
7              (= kinds-of-coins 0))
8          0)
9         (else
10          (+ (cc amount (- kinds-of-coins 1))
11              (cc (- amount (first-denomination
12                          kinds-of-coins))
13                  kinds-of-coins))))))
14
15 (define (first-denomination kinds-of-coins)
16   (cond ((= kinds-of-coins 1) 1)
17         ((= 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 illustrating the process generated by the `count-change` procedure of 1.2.2 in making change for 11 cents.

### 2.26.3 Answer

I want to generate this graph algorithmically.

```
1 ;; cursed global
2 (define bubblecounter 0)
3 ;; Returns # of ways change can be made
4 ;; "Helper" for (cc)
5 (define (count-change amount)
6   (display "digraph {\n"} ;; start graph
7   (cc amount 5 0)
8   (display "}\n") ;; end graph
9   (set! bubblecounter 0))
10
11 ;; GraphViz output
12 ;; Derivative:
13   ↪ https://stackoverflow.com/a/14806144
14 (define (cc amount kinds-of-coins oldbubble)
15   (let ((recur (lambda (new-amount new-kinds)
16                   (begin
17                     (display "\n") ;; Source
18                     ↪ bubble
19                     (display `(\,oldbubble
20                               ,amount ,kinds-of-coins))
21                     (display "\n")
22                     (display " -> ") ;; arrow
23                     ↪ pointing from parent to child
24                     (display "\n") ;; child
25                     ↪ bubble
26                     (display `(\,bubblecounter
27                               ,new-amount ,new-kinds))
28                     (display "\n")
29                     (display "\n")
30                     (cc new-amount new-kinds
31                         bubblecounter))))))
31   (set! bubblecounter (+ bubblecounter 1))
32   (cond ((= amount 0) 1)
```

```

27      ((or (< amount 0) (= kinds-of-coins
↪ 0)) 0)
28      (else (+
29              (recur amount (-
↪ kinds-of-coins 1))
30              (recur (- amount
31                      (first-denomination
↪ kinds-of-coins))
32                      kinds-of-coins))))))
33
34 (define (first-denomination kinds-of-coins)
35   (cond ((= kinds-of-coins 1) 1)
36         ((= kinds-of-coins 2) 5)
37         ((= kinds-of-coins 3) 10)
38         ((= kinds-of-coins 4) 25)
39         ((= kinds-of-coins 5) 50)))

```

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

```

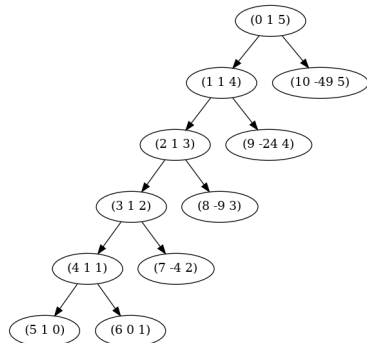
1 <<count-change-graphviz>>
2 (count-change 1)

```

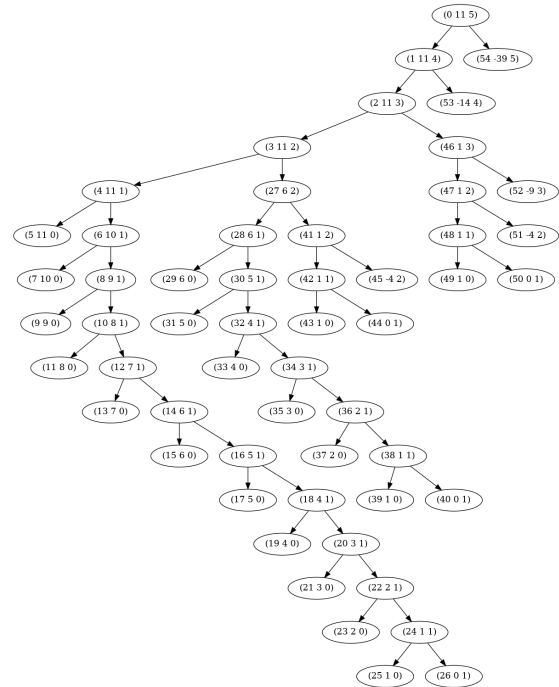
```

1 digraph {
2   "(0 1 5)" -> "(1 1 4)"
3   "(1 1 4)" -> "(2 1 3)"
4   "(2 1 3)" -> "(3 1 2)"
5   "(3 1 2)" -> "(4 1 1)"
6   "(4 1 1)" -> "(5 1 0)"
7   "(4 1 1)" -> "(6 0 1)"
8   "(3 1 2)" -> "(7 -4 2)"
9   "(2 1 3)" -> "(8 -9 3)"
10  "(1 1 4)" -> "(9 -24 4)"
11  "(0 1 5)" -> "(10 -49 5)"
12 }

```



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



## 2.26.4 Question B

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

## 2.26.5 Answer B

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

```

1 (define (count-calls amount)
2   (cc-calls amount 5))
3
4 (define (cc-calls amount kinds-of-coins)
5   (cond ((= amount 0) '(1 . 1))
6         ((or (< amount 0)
7              (= kinds-of-coins 0))
8           '(0 . 1))

```

```

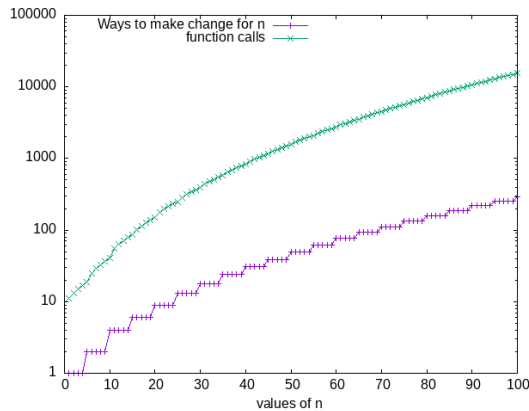
9      (else
10      (let ((a (cc-calls amount (-
↪ kinds-of-coins 1)))
11          (b (cc-calls (- amount
↪ (first-denomination
12      ↪ kinds-of-coins)))
13          kinds-of-coins)))
14      (cons (+ (car a)
15              (car b))
16            (+ 1
17              (cdr a)
18              (cdr b))))))
19
20 (define (first-denomination kinds-of-coins)
21   (cond ((= kinds-of-coins 1) 1)
22         ((= kinds-of-coins 2) 5)
23         ((= kinds-of-coins 3) 10)
24         ((= kinds-of-coins 4) 25)
25         ((= kinds-of-coins 5) 50)))

```

```

1 (use-srfis '(1))
2 <<cc-calls>>
3 (let* ((vals (drop (iota 101) 1))
4        (mine (map count-calls vals)))
5   (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  $\Theta$  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  $\Theta$  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 be bounded by lines of  $n^2 \times \text{someconstant}$ , 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  $\Theta$  and  $\Omega$  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  $\Theta$  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:

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

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

After seeing a few solutions that I found somewhat confusing, I landed on this awesome article from Codology.net. They show how you can remove the

summation, and proposed this equation for count-change with 5 denominations:

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

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

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

## 2.27 Exercise 1.15

### 2.27.1 Question A

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

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

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

### 2.27.2 Answer A

Let's find out!

```
1 (define (cube x) (* x x x))
2 (define (p x) (- (* 3 x) (* 4 (cube x))))
3 (define (sine angle)
```

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

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

-0.39980345741334 5

`p` is evaluated 5 times.

### 2.27.3 Question B

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

### 2.27.4 Answer B

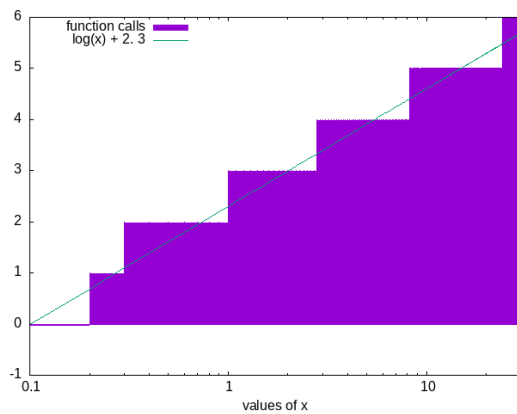
```
1 (use-srfis '(1))
2 <<1-15-p-measure>>
3 (let* ((vals (iota 300 0.1 0.1))
4         (sines (map (lambda (i)
5                       (cdr (sine i)))
6                       vals)))
7   (zip vals sines))
```

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

Example output:



	0.1	0
	0.2	1
0.30000000000000004	2	
	0.4	2
	0.5	2
	0.6	2
0.70000000000000001	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:

```

1 (sine 12.15)
2 (p (sine 4.05))
3 (p (p (sine 1.35)))
4 (p (p (p (sine 0.45))))
5 (p (p (p (p (sine 0.15)))))
6 (p (p (p (p (p (sine 0.05))))))
7 (p (p (p (p (p (p (sine 0.015)))))))
8 (p (p (p (p (p (p (p (sine 0.0045))))))))

```

```

9 (p (p (p 0.43513455050000005)))
10 (p (p 0.9758465331678772))
11 (p -0.7895631144708228)
12 -0.39980345741334

```

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

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

```

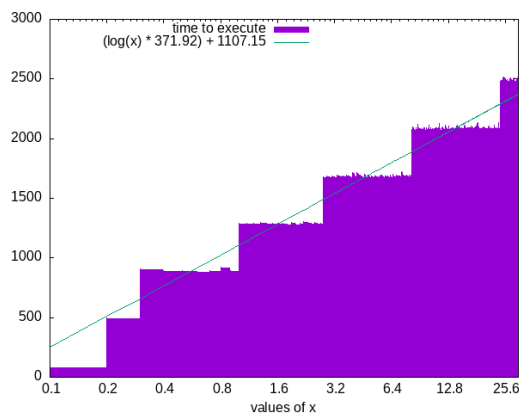
1 ;; This execution takes too long for org-mode,
  ↳ so I'm doing it
2 ;; externally and importing the results
3 (use-srfis '(1))
4 (use-modules (ice-9 format))
5 (load "../mattbench.scm")
6 <<1-15-deps>>
7 (let* ((vals (iota 300 0.1 0.1))
8         (times (map (lambda (i)
9                       (mattbench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

```

```

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

```



### 1. 1.2.4 Exponentiation

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

```
1 (define (expt b n)
2   (expt-iter b n 1))
3 (define (expt-iter b counter product)
4   (if (= counter 0)
5       product
6       (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:

$$\begin{aligned} b^2 &= b \cdot b \\ b^4 &= b^2 \cdot b^2 \\ b^8 &= b^4 \cdot b^4 \end{aligned}$$

## 2.28 Exercise 1.16

### 2.28.1 Text

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

### 2.28.2 Question

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

### 2.28.3 Diary

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

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

```

1 <<square>>
2 (define (fast-expt-iter b n)
3   (define (iter b n a)
4     (format #t "~&|~s~/~/|~s~/~/|~s|~%" b n a)
5     ↪ " ~&|~s~/~/|~s~/~/|~s|~%" (* b a) 1 1)
6     ((* b a)))
7     ((even? n) (iter (square b)
8                       (/ n 2)
9                       a))
10    (else (iter b (- n 1) (+ a 1))))))
11 (format #t "|~a~/|~a~/|~a|~%" "base" "power"
12 ↪ "variable")
13 (format #t "~&|--|--|~%"
14 (iter b n 1))

```

```

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

```

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

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

## 2.28.4 Answer

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

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

The second which I needed to deduce was this:

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

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  $a \rightarrow a \times b$

```

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

```

```

1 <<fast-expt-iter>>
2 <<try-these>>
3 (try-these (λ(x) (fast-expt-iter 3 x)) (cdr
4 ↪ (iota 11)))

```

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

## 2.29 Exercise 1.17

### 2.29.1 Question

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

```

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

```

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

## 2.29.2 Answer

```

1 (define (double x)
2   (+ x x))
3 (define (halve x)
4   (/ x 2))
5 (define (fast-mult-rec a b)
6   (cond ((= b 0) 0)
7         ((even? b)
8          (double (fast-mult-rec a (halve b))))
9         (<= My first instinct is iterative
10          (fast-mult (double a) (halve b)))
11         (else (+ a (fast-mult-rec a (- b
12   1))))))

```

Proof it works:

```

1 <<fast-mult-rec>>
2 <<try-these>>
3 (try-these (λ(x) (fast-mult-rec 3 x)) (cdr
4   (iota 11)))

```

```

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

```

## 2.30 Exercise 1.18

### 2.30.1 Question

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

### 2.30.2 Diary

1. Comparison benchmarks:

```

1 (load "../mattbench.scm")
2 <<fast-mult-iter>>
3 <<fast-mult-rec>>
4 <<print-table>>
5 (print-table (list (list "fast-mult-rec"
6   "fast-mult-iter")
7   (list (mattbench (λ()
8     (fast-mult-rec 32 32)) 10000000)
9     (mattbench (λ()
10      (fast-mult 32 32)) 10000000)))
11   #:colnames #t)

```

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

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

2. Hall of shame

Some of my *very* incorrect ideas:

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

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

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

### 2.30.3 Answer

```

1 (define (double x)
2   (+ x x))
3 (define (halve x)
4   (/ x 2))
5 (define (fast-mult a b)
6   (define (iter a b c)

```

```

7      (cond ((= b 0) 0)
8            ((= b 1) (+ a c))
9            ((even? b)
10             (iter (double a) (halve b) c))
11             (else (iter a (- b 1) (+ a c))))))
12      (iter a b 0))

```

```

1  <<fast-mult-iter>>
2  <<try-these>>
3  (try-these (λ(x) (fast-mult 3 x)) (cdr (iota
    ↪ 11))))

```

```

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

```

## 2.31 Exercise 1.19

### 2.31.1 Question

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

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

Call this transformation  $T$ , and observe that applying  $T$  over and over again  $n$  times, starting with 1 and 0, produces the pair  $\text{Fib}_{(n+1)}$  and  $\text{Fib}_{(n)}$ . In other words, the Fibonacci numbers are produced by applying  $T^n$ , the  $n$ th 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  $a \leftarrow -bq + aq + ap$  and  $b \leftarrow -bp + aq$ . 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:

```

1  (define (fib n)
2    (fib-iter 1 0 0 1 n))
3
4  (define (fib-iter a b p q count)
5    (cond ((= count 0) b)
6          ((even? count)
7           (fib-iter a
8                     b
9                     <??> ; compute p'
10                    <??> ; compute q'
11                    (/ count 2)))
12          (else (fib-iter (+ (* b q) (* a q) (*
    ↪ a p))
                           (+ (* b p) (* a q))
                           p
                           q
                           (- count 1)))))
13
14
15
16

```

### 2.31.2 Diary

More succinctly put:

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

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

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

```

1 (define (T p q)
2   (λ (a b)
3     (cons (+ (* b q) (* a q) (* a p))
4           (+ (* b p) (* a q)))))
5
6 (define T-fib
7   (T 0 1))
8
9 ;; Repeatedly apply T functions:
10 (define (Tr f n)
11   (Tr-iter f n 0 1))
12 (define (Tr-iter f n a b)
13   (if (= n 0)
14       a
15       (let ((l (f a b)))
16         (Tr-iter f (- n 1) (car l) (cdr l)))))

```

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

```

1 <<T-func>>
2 <<try-these>>
3 (try-these (λ (x) (Tr (T 0 1) x)) (cdr (iota
4   ↪ 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

```

1 (define (fib-rec n)
2   (cond ((= n 0) 0)
3         ((= n 1) 1)
4         (else (+ (fib-rec (- n 1))
5                  (fib-rec (- n 2))))))
6 (define (fib n)
7   (fib-iter 1 0 0 1 n))

```

```

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

```

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

## 2.32 1.2.5: Greatest Common Divisor

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

$$a \% b = r$$

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

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  $k^{\text{th}}$  Fibonacci number.

## 2.33 Exercise 1.20

### 2.33.1 Text

```

1 (define (gcd a b)
2   (if (= b 0)
3       a
4       (gcd b (remainder a b))))

```

### 2.33.2 Question

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

### 2.33.3 Answer

I struggled to understand this, but the key here is that normal-order evaluation causes the unevaluated

expressions to be duplicated, meaning they get evaluated multiple times.

#### 1. Applicative order

```

1 call (gcd 206 40)
2 (if)
3 (gcd 40 (remainder 206 40))
4 eval remainder before call
5 call (gcd 40 6)
6 (if)
7 (gcd 6 (remainder 40 6))
8 eval remainder before call
9 call (gcd 6 4)
10 (if)
11 (gcd 2 (remainder 4 2))
12 eval remainder before call
13 call (gcd 2 0)
14 (if)
15 ;; => 2

```

```

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

```

```

26      (remainder 40 (remainder 206 40)))
27
28 ; encounter conditional
29 (if (= (remainder 40 (remainder 206 40))
30      ↪ 0)
31      (remainder 206 40)
32      (gcd (remainder 40 (remainder 206 40))
33           ↪ (remainder (remainder 206 40)
34                ↪ (remainder 40 (remainder 206 40)))))
35
36 ; eval 2 remainder
37 (if (= 4 0)
38     (remainder 206 40)
39     (gcd (remainder 40 (remainder 206 40))
40         ↪ (remainder (remainder 206 40)
41              ↪ (remainder 40 (remainder 206 40)))))
42
43 ; recurse
44 (gcd (remainder 40 (remainder 206 40))
45     ↪ (remainder (remainder 206 40)
46          ↪ (remainder 40 (remainder 206 40)))))
47
48 ; encounter conditional
49 (if (= (remainder (remainder 206 40)
50     ↪ (remainder 40 (remainder 206 40))) 0)
51     (remainder 40 (remainder 206 40))
52     (gcd (remainder (remainder 206 40)
53         ↪ (remainder 40 (remainder 206 40))
54         ↪ (remainder (remainder 40
55             ↪ (remainder 206 40)) (remainder
56             ↪ (remainder 206 40) (remainder 40
57             ↪ (remainder 206 40)))))
58
59 ; eval 4 remainders
60 (if (= 2 0)
61     (remainder 40 (remainder 206 40))
62     (gcd (remainder (remainder 206 40)
63         ↪ (remainder 40 (remainder 206 40))
64         ↪ (remainder (remainder 40
65             ↪ (remainder 206 40)) (remainder
66             ↪ (remainder 206 40) (remainder 40
67             ↪ (remainder 206 40)))))
68
69 ; eval 7 remainders
70 (if (= 0 0)
71     (remainder (remainder 206 40)
72     ↪ (remainder 40 (remainder 206 40))
73     (gcd (remainder (remainder 40
74         ↪ (remainder 206 40)) (remainder
75         ↪ (remainder 206 40) (remainder 40
76         ↪ (remainder 206 40)) (remainder a
77         ↪ (remainder (remainder 40 (remainder
78             ↪ 206 40)) (remainder (remainder 206
79             ↪ 40) (remainder 40 (remainder 206
80             ↪ 40)))))
81
82 ; eval 4 remainders
83 (remainder (remainder 206 40) (remainder
84     ↪ 40 (remainder 206 40)))
85
86 ; => 2

```

```

58      (remainder (remainder 40 (remainder
59     ↪ 206 40)) (remainder (remainder 206
60     ↪ 40) (remainder 40 (remainder 206
61     ↪ 40)))))
62
63 ; encounter conditional
64 (if (= (remainder (remainder 40
65     ↪ (remainder 206 40)) (remainder
66     ↪ (remainder 206 40) (remainder 40
67     ↪ (remainder 206 40))))) 0)
68     (remainder (remainder 206 40)
69     ↪ (remainder 40 (remainder 206 40)))
70     (gcd (remainder (remainder 40
71         ↪ (remainder 206 40)) (remainder
72         ↪ (remainder 206 40) (remainder 40
73         ↪ (remainder 206 40)) (remainder a
74         ↪ (remainder (remainder 40 (remainder
75             ↪ 206 40)) (remainder (remainder 206
76             ↪ 40) (remainder 40 (remainder 206
77             ↪ 40)))))
78
79 ; eval 7 remainders
80 (if (= 0 0)
81     (remainder (remainder 206 40)
82     ↪ (remainder 40 (remainder 206 40))
83     (gcd (remainder (remainder 40
84         ↪ (remainder 206 40)) (remainder
85         ↪ (remainder 206 40) (remainder 40
86         ↪ (remainder 206 40)) (remainder a
87         ↪ (remainder (remainder 40 (remainder
88             ↪ 206 40)) (remainder (remainder 206
89             ↪ 40) (remainder 40 (remainder 206
90             ↪ 40)))))
91
92 ; eval 4 remainders
93 (remainder (remainder 206 40) (remainder
94     ↪ 40 (remainder 206 40)))
95
96 ; => 2

```

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

## 2.34 1.2.6: Example: Testing for Primality

Two algorithms for testing primality of numbers.

1.  $\Theta(\sqrt{n})$ : Start with  $x = 2$ , check for divisibility



with  $n$ , if not then increment  $x$  by 1 and check again. If  $x^2 > n$  and you haven't found a divisor,  $n$  is prime.

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

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

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

## 2.35 Exercise 1.21

### 2.35.1 Text

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

### 2.35.2 Question

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

### 2.35.3 Answer

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

199 1999 7

## 2.36 Exercise 1.22

### 2.36.1 Question

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

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

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

```

14 (display " *** ")
15 (display elapsed-time))

```

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

## 2.36.2 Answer

### 1. Part 1

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

```

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

```

```

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

```

```

1 1001
2 1003
3 1005
4 1007
5 1009 *** 1651
6 1011
7 1013 *** 1425
8 1015
9 1017
10 1019 *** 1375

```

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

```

1 <<search-primes-basic>>
2 (let ((lt1000-1 (search-for-primes 1000
3   ↪ 3))
4       (lt10000-1 (search-for-primes 10000
5   ↪ 3))
6       (lt100000-1 (search-for-primes 100000
7   ↪ 3))
8       (lt1000000-1 (search-for-primes 1000000
9   ↪ 3)))
10 (list
11   (list "Primes > 1000" (reverse lt1000-1))
12   (list "Primes > 10000" (reverse lt10000-1))
13   (list "Primes > 100000" (reverse lt100000-1))
14   (list "Primes > 1000000" (reverse lt1000000-1))
15 ))

```

Primes > 1000	(1009 1013 1019)
Primes > 10000	(10007 10009 10037)
Primes > 100000	(100003 100019 100043)
Primes > 1000000	(1000003 1000033 1000037)

### 2. Part 2

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

```

1 (define (mattbench2 f n)
2   ;; Executes "f" for n times, and
3   ↪ returns how long it took.
4   ;; f is a lambda that takes no
5   ↪ arguments, a.k.a. a "thunk"
6
7   ;; Returns a list with car(last
8   ↪ execution results) and cadr(time
9   ↪ taken divided by iterations n)
10
11   (define (time-getter)
12     ↪ (get-internal-run-time))
13   (define start-time (time-getter))
14   (define (how-long) (- (time-getter)
15     ↪ start-time))
16
17   (define (iter i)
18     (f)
19     (if (<= i 0)
20       (f) ;; return the results of the
21       ↪ last function call
22       (iter (- i 1))))
23
24   (list (iter n) ;; result of last call
25     ↪ of f
26     (/ (how-long) (* n 1.0))));
27   ↪ Divide by iterations so changed n has
28   ↪ no effect

```

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

```

1 (define (search-for-primes minimum goal)
2   (define m (if (even? minimum)
3     (+ minimum 1)
4     (minimum)))
5   (search-for-primes-iter m '() goal))
6 (define (search-for-primes-iter n lst
7   ↪ goal)
8   (if (= goal 0)

```

```

8     lst
9     (let ((x (prime? n)))
10      (if (not (equal? x #f))
11        (search-for-primes-iter (+ n
12    ↪ 2) (cons n lst) (- goal 1))
13        (search-for-primes-iter (+ n
14    ↪ 2) lst goal))))))

```

I can benchmark these functions like so:

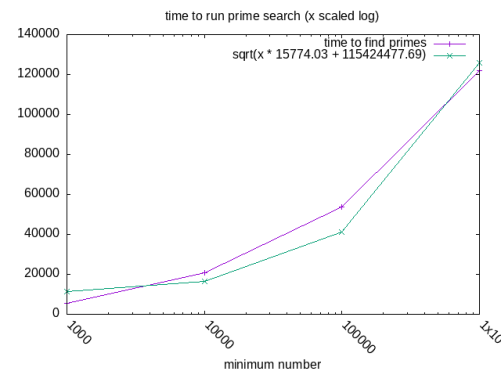
```

1 <<mattbench2>>
2 <<prime-smallest-divisor>>
3 <<search-for-primes-untimed>>
4 <<print-table>>
5
6 (define benchmark-iterations 1000000)
7 (define (testit f)
8   (list (cadr (mattbench2 (λ() (f 1000
9     ↪ 3)) benchmark-iterations))
10     (cadr (mattbench2 (λ() (f 10000
11     ↪ 3)) benchmark-iterations))
12     (cadr (mattbench2 (λ() (f 100000
13     ↪ 3)) benchmark-iterations))
14     (cadr (mattbench2 (λ() (f 1000000
15     ↪ 3)) benchmark-iterations))))
16
17 (print-row
18   (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  $\Theta$  notation I'm pretty clueless;

## 2.37 Exercise 1.23

### 2.37.1 Question

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

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

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

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

(define benchmark-iterations 1000000)
(define (testit f)
  (list (cadr (mattbench2 (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))
```

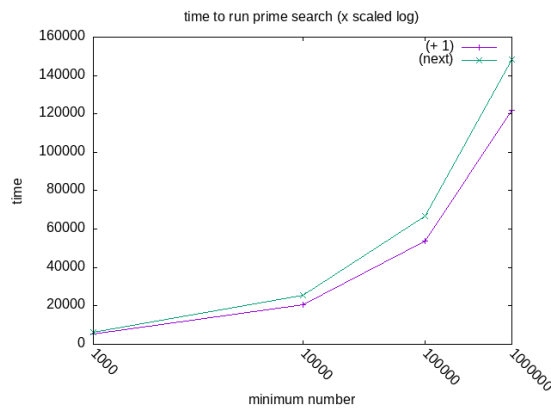
### 2.37.2 A Comedy of Error (just the one)

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

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

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.

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

```
1 <<mattbench2>>
2 <<find-divisor-faster-real>>
3 (define (prime? n)
4   (= n (smallest-divisor n)))
5 <<search-for-primes-untimed>>
6 <<print-table>>
7
8 (define benchmark-iterations 500000)
9 (define (testit f)
10  (list (cadr (mattbench2 (λ() (f 1000 3)))
11        benchmark-iterations))
12        (cadr (mattbench2 (λ() (f 10000 3)))
13              benchmark-iterations))
14        (cadr (mattbench2 (λ() (f 100000 3)))
15              benchmark-iterations))
16        (cadr (mattbench2 (λ() (f 1000000 3)))
17              benchmark-iterations))))
18
19 (print-row
20  (testit search-for-primes))
```

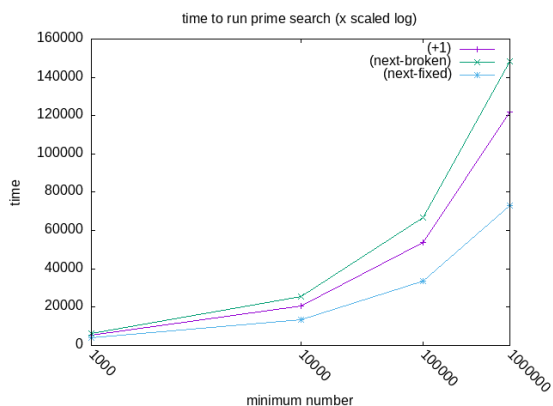
3863.7424 13519.209814 33520.676384 73005.539932

### 2.37.3 Answer

Ok, let's try that again.

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

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:

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

Using it for percentage comparisons:

```
1 <<average-varargs>>
2 (list (cons "% speedup for broken (next)"
3           (cons (format #f "~2$"
4                       (apply average
5                           (map (lambda (x y)
6                                (* 100 (/ x y)))
7                               (car
8                                 (smd) (car smdff))))
9                               #nil))
10          (cons "% speedup for real (next)"
11                (cons (format #f "~2$"
12                            (apply average
13                                (map (lambda (x y)
14                                     (* 100 (/ x y)))
15                                    (car
16                                      (smd) (car smdff))))
17                                #nil))))
```

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

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

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

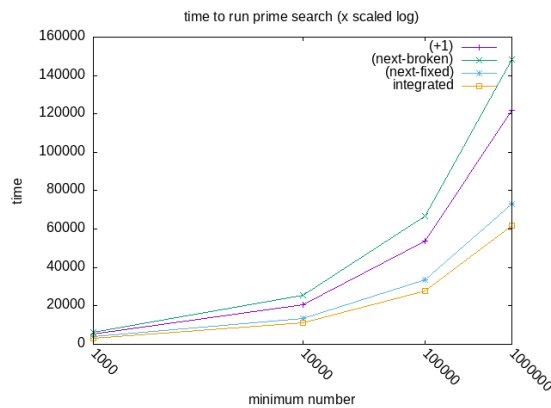
```
1 <<square>>
2 (define (smallest-divisor n)
3   (if (divides? 2 n)
4       2
5       (find-divisor n 3))) ;; check
```

```
5 (find-divisor n 3))) ;; start
6 ↪ find-divisor at 3
7
8 (define (find-divisor n test-divisor)
9   (cond ((> (square test-divisor) n)
10          n)
11         ((divides? test-divisor n)
12          test-divisor)
13         (else (find-divisor
14                 n
15                 (+ 2 test-divisor)))))) ;; just
16 ↪ increase by 2
17
18 (define (divides? a b)
19   (= (remainder b a) 0))
```

```
1 <<mattbench2>>
2 <<find-divisor-faster-real2>>
3 (define (prime? n)
4   (= n (smallest-divisor n)))
5 <<search-for-primes-untimed>>
6 <<print-table>>
7
8 (define benchmark-iterations 500000)
9 (define (testit f)
10  (list (cadr (mattbench2 (lambda () (f 1000 3)))
11        (cadr (mattbench2 (lambda () (f 10000 3)))
12        (cadr (mattbench2 (lambda () (f 100000 3)))
13        (cadr (mattbench2 (lambda () (f 1000000 3)))
14        (cadr (mattbench2 (lambda () (f 10000000 3))))))
15  (print-row
16    (testit search-for-primes)))
```

```
3151.259574 11245.20428 27803.067944 61997.275154
```

	min	(+1)	(next-broken)	(next-fixed)	i
1000	—	—	—	—	3151.259574
10000	5425.223086	6456.538118	3863.7424	11245.20428	11245.20428
100000	20772.332491	25550.757304	13519.209814	27803.067944	27803.067944
1000000	53577.240193	66746.041644	33520.676384	61997.275154	61997.275154
10000000	121986.712395	148505.580638	73005.539932	3151.259574	3151.259574



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

## 2.38 Exercise 1.24

### 2.38.1 Text

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

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

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

### 2.38.2 Question

Modify the timed-prime-test procedure of 2.36 to use fast-prime? (the Fermat method), and test each of the 12 primes you found in that exercise. Since the Fermat test has  $\Theta(\log n)$  growth, how would you expect the time to test primes near 1,000,000 to compare with the time needed to test primes near 1000? Do your data bear this out? Can you explain any discrepancy you find?

### 2.38.3 Answer

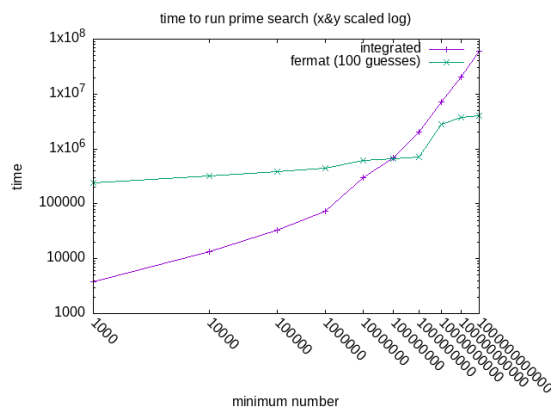
```
1 <<mattbench2>>
2 <<expmod>>
3 <<fermat-test>>
4 <<fast-prime>>
5 (define fermat-iterations 2)
6 (define (prime? n)
7   (fast-prime? n fermat-iterations))
8 <<search-for-primes-untimed>>
9 <<print-table>>
10
11 (define benchmark-iterations 500000)
12 (define (testit f)
13   (list (cadr (mattbench2 (λ() (f 1000 3)))
14         ↪ benchmark-iterations))
15         (cadr (mattbench2 (λ() (f 10000 3)))
16         ↪ benchmark-iterations))
17         (cadr (mattbench2 (λ() (f 100000 3)))
18         ↪ benchmark-iterations))
19         (cadr (mattbench2 (λ() (f 1000000 3)))
20         ↪ benchmark-iterations))))
21
22 (print-row
23   (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







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

## 2.39 Exercise 1.25

### 2.39.1 Question

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

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

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

### 2.39.2 Answer

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

remainder after every step, significantly reducing the memory requirements.

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

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

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

## 2.40 Exercise 1.26

### 2.40.1 Question

Louis Reasoner is having great difficulty doing Exercise 1.24. His `fast-prime?` test seems to run more slowly than his `prime?` test. Louis calls his friend Eva Lu Ator over

to help. When they examine Louis's code, they find that he has rewritten the `expmod` procedure to use an explicit multiplication, rather than calling `square`:

```

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

```

↪ hmm.

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

## 2.40.2 Answer

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

## 2.41 Exercise 1.27

### 2.41.1 Question

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

561   1105   1729   2465   2821   6601

### 2.41.2 Answer

```

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

```

```

1 <<car-test>>
2 (list (car-test 12) ; <== false (not prime)
3       (car-test 1009); <== true  (real prime)
4       (car-test 561)); <== true  (not prime,
5                               Carmichael number)

```

## 2.42 Exercise 1.28

### 2.42.1 Question

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

ial square root of 1, and use this to implement the Miller-Rabin test with a procedure analogous to `fermat-test`. Check your procedure by testing various known primes and non-primes. Hint: One convenient way to make `expmod` signal is to have it return 0.

## 2.42.2 Analysis

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

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

```
1 ;; fermt prime test but checks *every* value
2 ↪ from 2 to n-1
3 (define (fermat-prime? n)
4   (define (iter a)
5     (if (= a n)
6         #f
7         (if (= (expmod a n n) a)
8             #t
9             (iter (+ 1 a)))))
10   (iter 2))
```

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

```
port)))
```

This will be useful in various future functions:

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

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

(prime?) is working: Yes

(fermat-prime?) is vulnerable: Yes

## 2.42.3 Answer

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

```

8         0
9         (remainder sqr m)))
10     (else
11       (remainder
12        (* base (expmod-mr base (- exp 1) m))
13         m))))

```

```

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

```

```

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

```

```

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

```

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

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

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

```

1 <<square>>
2 (define (expmod-mr base exp m)
3   (cond ((= exp 0) 1)
4         ((even? exp)
5          ;; Keep result and remainder separate

```

```

6         (let* ((result (expmod-mr base (/ exp
7   ↪ 2) m))
8               (rem (remainder (square
9   ↪ result) m)))
10          (if (and (not (= result 1))
11                  (not (= result (- m 1)))
12                  (= 1 rem))
13              0 ;; non-trivial sqrt mod 1 is
14              ↪ found
15              rem)))
16          (else
17            (remainder
18             (* base (expmod-mr base (- exp 1) m))
19             m))))

```

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

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

```

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

```

One more time.

```

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

```

```

1 <<expmod-mr2>>
2 <<mr-test2>>
3 <<mr-prime>>
4 (define mr-times 100)
5 <<get-lists-of-primes>>
6 (format #t |
7   ↪ "      mr detects primes: ~a~%mr false-positives Carmichael
8     (and-map (λ(x)(mr-prime? x mr-times))
9     ↪ list-of-primes)
10    (and-map (λ(x)(mr-prime? x mr-times))
11    ↪ list-of-carmichaels))

```

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

## 2.43 1.3: Formulating Abstractions with Higher-Order Procedures

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

### 2.44 1.3.1: Procedures as Arguments

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

For the program:

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

Which is equivalent to:

$$\sum_{n=a}^b \text{term}(n) = \text{term}(a) + \text{term}(\text{next}(a)) + \text{term}(\text{next}(\text{next}(a))) + \dots + \text{term}(b)$$

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

## 2.45 Exercise 1.29

### 2.45.1 Text

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

```
1 (define (integral f a b dx)
2   (define (add-dx x)
3     (+ x dx))
```

```
4 (* (sum f (+ a (/ dx 2.0)) add-dx b)
5    dx))
```

### 2.45.2 Question

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

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

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

### 2.45.3 Answer

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

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

```

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

```

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

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

## 2.46 Exercise 1.30

### 2.46.1 Question

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

### 2.46.2 Answer

```

1 (define (sum-iter term a next b)
2   (define (iter a result)

```

```

3     (if (> a b)
4         result
5         (iter (next a) (+ result (term a)))))
6   (iter a 0))

```

Let's check the stats!

recursive	iterative
30051.080005	19568.685587

## 2.47 Exercise 1.31

### 2.47.1 Question A.1

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

### 2.47.2 Answer A.1

```

1 (define (product-iter term a next b)
2   (define (iter a result)
3     (if (> a b)
4         result
5         (iter (next a) (* result (term a)))))
6   (iter a 1)) ;; start at 1 so it's not always
7   ↪ 0

```

### 2.47.3 Question A.2

Show how to define factorial in terms of product.

### 2.47.4 Answer A.2

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

```

1 <<product-iter>>
2 (define (identity x)
3   x)

```

```

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

```

### 2.47.5 Question A.3

Also use product to compute approximations to  $\pi$  using the formula

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

### 2.47.6 Answer A.3

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

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

```

1 <<product-iter>>
2 (define (pi-product n)
3   (define (div x)
4     (let ((x1 (- x 1))
5           (x2 (+ x 1)))
6       (* (/ x x1) (/ x x2))))
7   (* 2.0 (product-iter div 2 (lambda (z) (+ z
8     ↪ 2)) n)))
9 (display (pi-product 100000))

```

3.1415769458228726

### 2.47.7 Question B

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

### 2.47.8 Answer B

```

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

```

```

1 <<mattbench2>>
2 <<print-table>>
3 <<product-iter>>
4 (define (pi-product n)
5   (define (div x)
6     (let ((x1 (- x 1))
7           (x2 (+ x 1)))
8       (* (/ x x1) (/ x x2))))
9   (* 2.0 (product-iter div 2 (lambda (z) (+ z
10     ↪ 2)) n)))
11 <<product-rec>>
12 (define (pi-product-rec n)
13   (define (div x)
14     (let ((x1 (- x 1))
15           (x2 (+ x 1)))
16       (* (/ x x1) (/ x x2))))
17   (* 2.0 (product-rec div 2 (lambda (z) (+ z
18     ↪ 2)) n)))
19 (define iterations 50000)
20 (print-table
21  (list (list "iterative" "recursive")
22        (list (cadr (mattbench2 (lambda () (pi-product
23     ↪ 1000)) iterations))
24              (cadr (mattbench2
25     ↪ (lambda () (pi-product-rec 1000)) iterations))))
26  #:colnames #t)

```

iterative	recursive
1267118.0538	3067085.5323

## 2.48 Exercise 1.32

### 2.48.1 Question A

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

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

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

### 2.48.2 Answer A

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

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

```
1 <<accumulate-iter>>
2
3 ;; here you can see definitions in terms of
4 ↪ accumulate
5 (define (sum term a next b)
6   (accumulate-iter + 0 term a next b))
7 (define (product term a next b)
8   (accumulate-iter * 1 term a next b))
9
10 (define (identity x)
11   x)
12 (define (inc x)
13   (1+ x))
14
15 ;; accumulate in action
16 (define (factorial n)
17   (accumulate-iter * 1 identity 1 inc n))
```

```
17 (display (factorial 7))
18
```

5040

### 2.48.3 Question B

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

### 2.48.4 Answer B

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

## 2.49 Exercise 1.33

### 2.49.1 Question A

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

### 2.49.2 Answer A

```
1 (define (filtered-accumulate-iter
2   predicate? combiner null-value
3   term a next b)
4   (define (iter a result)
5     (cond ((> a b) result)
6           ((predicate? a)
7            (combiner result (term a)
8                          (iter (next a) next
9                              predicate? combiner null-value
10                              term (next a) next
11                              ↪ b))))))
```



```

7      (iter (next a)
8            (combiner result (term a))))
9      (else (iter (next a)
10                  result))))
11 (iter a null-value))

```

### 2.49.3 Question B

Show how to express the following using filtered-accumulate:

1. A

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

```

1 (load "mattcheck.scm")
2 (define (square x)
3   (* x x))
4 <<filtered-accumulate-iter>>
5 <<expmod-mr2>>
6 <<mr-test2>>
7 <<mr-prime>>
8 (define mr-times 100)
9 (define (prime? x)
10   (mr-prime? x mr-times))
11 (define (prime-sum a b)
12   (filtered-accumulate-iter prime? + 0
13                             square a 1+
14                             → b))
15 (mattcheck-equal "1 prime correct"
16   (prime-sum 1008 1010)
17   (square 1009)) ;; 1009
18 (mattcheck-equal "many primes correct"
19   (prime-sum 1000 2001)
20   (apply +
21     (map square
22       (filter
23         → prime? (iota (- 2001 1000)
24         → 1000))))))

```

SUCCEED at 1 prime correct

SUCCEED at many primes correct

2. B

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

```

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

```

SUCCEED at Ex\_1-33B

## 2.50 1.3.2: Constructing Procedures Using lambda

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

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

### 2.51 Exercise 1.34

#### 2.51.1 Question

Suppose we define the procedure

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

Then, we have

```
1 (f square)
2 ; 4
3 (f (lambda (z) (* z (+ z 1))))
4 ; 6
```

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

### 2.51.2 Answer

It ends up trying to execute 2 as a function.

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

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

equivalent to `(lambda (x) (+ x x))`. In English, “the function whose value at  $y$  is  $x/y$ ”. Though it seems like  $\mapsto$  doesn’t necessarily describe a function, but the value of a function at a certain point? Or maybe that would just be , ie  $f(x)$  etc

## 2.53 Exercise 1.35

### 2.53.1 Text

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

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

## 2.52 1.3.3 Procedures as General Methods

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

### 2.53.2 Question

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

### 2.53.3 Answer

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

1.6180327868852458

## 2.54 Exercise 1.36

### 2.54.1 Question

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

### 2.54.2 Answer

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

```
1 (use-modules (ice-9 format))
2 (define tolerance 0.00001)
3
4 (define (fixed-point f first-guess)
5   (define (close-enough? v1 v2)
6     (< (abs (- v1 v2))
7       tolerance))
8   (define (try guess)
9     (let ((next (f guess)))
10      (format #t "~&a~%" next)
11      (if (close-enough? guess next)
12          next
13          (try next))))
14   (try first-guess))
```

```
1 <<close-enough>>
2 <<fixed-point-debug>>
3 (fixed-point (lambda (x) (/ (log 1000) (log x)))) 1.1)
```

Undamped, `fixed-point` makes 37 guesses.

```
1 <<close-enough>>
2 <<fixed-point-debug>>
3 (define (average x y)
4   (/ (+ x y) 2))
```

```
5 (fixed-point (lambda (x) (average (log x) (/ (log
  ↪ 1000) (log x)))) 1.1)
```

Damped, it makes 21.

## 2.55 Exercise 1.37

### 2.55.1 Question A

An infinite continued fraction is an expression of the form

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

As an example, one can show that the infinite continued fraction expansion with the  $N_i$  and the  $D_i$  all equal to 1 produces  $1/\varphi$ , where  $\varphi$  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

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

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

### 2.55.2 Answer A

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

```

1 (define (cont-frac n d k)
2   (define (iter i result)
3     (if (= i 0)
4         result
5         (iter (1- i) (/ (n i) (+ (d i)
6                               ↪ result)))))
7   (iter (1- k) (/ (n k) (d k))))

```

### 2.55.3 Question B

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

```

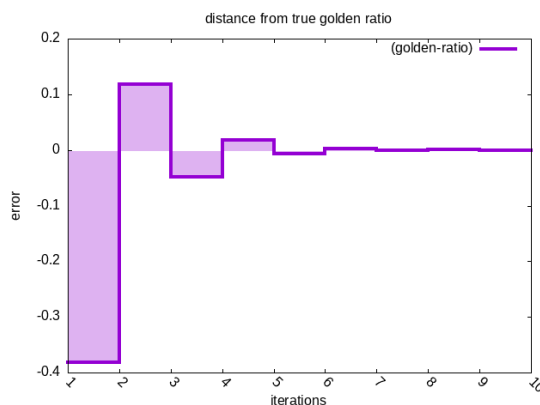
1 (cont-frac (lambda (i) 1.0)
2            (lambda (i) 1.0)
3            k)

```

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

### 2.55.4 Answer B

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



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

### 2.55.5 Question C

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

### 2.55.6 Answer C

```

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

```

```

1 <<cont-frac>>
2 <<cont-frac-rec>>
3 (define (golden-ratio k)
4   (cont-frac (lambda (i) 1.0) (lambda (i) 1.0) k))
5 (define (golden-ratio-rec k)
6   (cont-frac-rec (lambda (i) 1.0) (lambda (i) 1.0) k))
7
8 (load "mattcheck.scm")
9 (mattcheck-equal
10  ↪ "cont-frac iter and recursive equivalence"
11    (golden-ratio-rec 15)
12    (golden-ratio 15))

```

SUCCEED at cont-frac iter and recursive equivalence

## 2.56 Exercise 1.38

### 2.56.1 Question

In 1737, the Swiss mathematician Leonhard Euler published a memoir *De Fractionibus Continuis*, which included a continued fraction expansion for  $e - 2$ , where  $e$  is the base of the natural logarithms. In this fraction, the  $N_i$  are all 1, and the  $D_i$  are successively 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, ... Write

a program that uses your `cont-frac` procedure from Exercise 1.37 to approximate  $e$ , based on Euler's expansion.

## 2.56.2 Answer

```
1 <<cont-frac>>
2 (define (euler k)
3   (+ 2
4     (cont-frac (λ(i) 1.0)
5                 (λ(i) (let ((j (1+ i)))
6                       (if (= 0 (modulo j 3))
7                           (* 2 (/ j 3))
8                           1))))
9     k)))
10
11 (euler 100)
```

2.7182818284590455

## 2.57 Exercise 1.39

### 2.57.1 Question

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

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

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

### 2.57.2 Answer

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

```
6 (λ(i) (if (= i 1)
7           1.0
8           (- (* i 2.0) 1.0)))
9 k))
10
11 (tan-cf 55 101)
```

-45.1830879105221

## 2.58 1.3.4 Procedures as Returned Values

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

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

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

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

## 2.59 Exercise 1.40

### 2.59.1 Text

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

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

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

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

```

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

```

### 2.59.2 Question

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

```

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

```

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

### 2.59.3 Answer

```

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

```

```

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

```

```

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

```

## 2.60 Exercise 1.41

### 2.60.1 Question

Define a procedure `double` that takes a procedure of one argument as argument and returns a procedure that applies the original procedure twice. For example, if `inc` is

a procedure that adds 1 to its argument, then `(double inc)` should be a procedure that adds 2. What value is returned by

```

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

```

### 2.60.2 Answer

```

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

```

```

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

```

21

## 2.61 Exercise 1.42

### 2.61.1 Question

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

### 2.61.2 Answer

```

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

```

```

1 <<compose>>
2 <<square>>
3 (define inc 1+)
4 ((compose square inc) 6)

```

49

## 2.62 Exercise 1.43

### 2.62.1 Question

If  $f$  is a numerical function and  $n$  is a positive integer, then we can form the  $n^{\text{th}}$  repeated application of  $f$ , which is defined to be the function whose value at  $x$  is  $f(f(\dots(f(x))\dots))$ . For example, if  $f$  is the function  $x \mapsto x + 1$ , then the  $n^{\text{th}}$  repeated application of  $f$  is the function  $x \mapsto x + n$ . If  $f$  is the operation of squaring a number, then the  $n^{\text{th}}$  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  $n^{\text{th}}$  repeated application of  $f$ .

### 2.62.2 Answer

```
1 <<compose>>
2 (define (repeated f n)
3   (if (= n 1)
4       f
5       (repeated (compose f f)
6                 (- n 1))))
```

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

Success

## 2.63 Exercise 1.44

### 2.63.1 Question

The idea of smoothing a function is an important concept in signal processing. If  $f$  is a function and  $dx$  is some small number, then the smoothed version of  $f$  is the function whose value at a point  $x$  is the average of  $f(x - dx)$ ,  $f(x)$ , and  $f(x + dx)$ . Write a

procedure `smooth` that takes as input a procedure that computes  $f$  and returns a procedure that computes the smoothed  $f$ . It is sometimes valuable to repeatedly smooth a function (that is, smooth the smoothed function, and so on) to obtain the  $n$ -fold smoothed function. Show how to generate the  $n$ -fold smoothed function of any given function using `smooth` and `repeated` from Exercise 1.43.

### 2.63.2 Answer

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

## 2.64 Exercise 1.45

### 2.64.1 Question

We saw in 1.3.3 that attempting to compute square roots by naively finding a fixed point of  $y \mapsto x/y$  does not converge, and that this can be fixed by average damping. The same method works for finding cube roots as fixed points of the average-damped  $y \mapsto x/y^2$ . Unfortunately, the process does not work for fourth roots—a single average damp is not enough to make a fixed-point search for  $y \mapsto x/y^3$  converge. On the other hand, if we average damp twice (i.e., use the average damp of the average damp of  $y \mapsto x/y^3$ ) the fixed-point search does converge. Do some experiments to determine how many average damps are required to compute  $n^{\text{th}}$  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  $n^{\text{th}}$  roots using `fixed-point`, `average-damp`, and the `repeated` procedure

of Exercise 1.43. Assume that any arithmetic operations you need are available as primitives.

## 2.64.2 Answer

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

```
1  <<fixed-point-txt>>
2  <<repeated>>
3  <<average-damp>>
4  (define (sqrt n)
5    (fixed-point
6      (average-damp
7        (lambda (y)
8          (/ x y)))
9      1.0))
10 (define (nth-root x n)
11   (fixed-point
12     ((repeated average-damp (ceiling (sqrt n)))
13      (lambda (y)
14        (/ x (expt y (- n 1)))))
15     1.0))
```

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

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

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

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

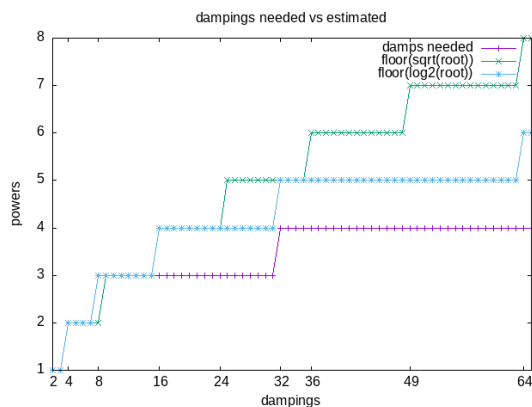
```
1  <<fixed-point-txt>>
2  <<limited-fixed-point>>
3  <<repeated>>
4  <<average-damp>>
5  <<average>>
6  <<print-table>>
7  (define (sqrt x)
8    (fixed-point
9      (average-damp
10        (lambda (y) (/ x y)))
11      1.0))
12 (define (nth-tester base n-max)
13   (define (iter ll)
14     (let ((n (+ 2 (length ll))))
15       (define (try damp)
16         (let ((x (limited-fixed-point
17                     ((repeated average-damp
18                               ↪ damp)
19                        (lambda (y)
20                          (/ base (expt y (- n
21                               ↪ 1))))))
22                     1.1)))
23         (if (string? x)
24             (try (1+ damp))
```



```

23         (list base n x damps))))
24     (if (> n n-max)
25         ll
26         (iter (cons (try 1) ll))))
27
28     (iter '())
29 (let* ((t (reverse (nth-tester 3 65))))
30     (cons '("root" "result" "damps needed"
31         ↪ "floor(sqrt(root))" "floor(log2(root))")
32         (map (λ(x)
33             (append x
34                 (list (floor (sqrt (car
35         ↪ x))))
36                 (list (floor (/ (log
37         ↪ (car x))(log 2))))))
38             (map cdr t))))

```



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

## 2.65 Exercise 1.46

### 2.65.1 Question

Several of the numerical methods described in this chapter are instances of an extremely general computational strategy known as *iterative improvement*. It-

erative 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

```

1 (define (iterative-improve good-enough?
2   ↪ improve)
3   (λ(first-guess)
4     (define (iter guess)
5       (let ((next (improve guess)))
6         (if (good-enough? guess next)
7             next
8             (iter next))))
9     (iter first-guess)))

```

```

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

```

```

1 <<average>>
2 <<iterative-improve>>
3 (define (improve guess x)
4   (average guess (/ x guess)))
5 (define (good-enough? guess x)
6   (= (improve guess x) guess))

```

```

7 (define (sqrt-improve x)
8   ((iterative-improve
9     (λ(guess) (improve guess x))
10    (λ(guess) (good-enough? guess x))))
11   1.0))

```

```

1 (load "mattcheck2.scm")
2 <<fixed-point-txt>>
3 <<fixed-point-improve>>
4 (mattcheck "fixed-point-improve still working"
5   (fixed-point (λ(x)(+ 1 (/ 1
6     ↪ x)))) 1.0)
7   (fixed-point-improve (λ(x)(+
8     ↪ 1 (/ 1 x)))) 1.0))
9 <<sqr>>
10 <<sqr-improve>>
11 (mattcheck "sqrt-improve still working"
    (sqrt 5)
    (sqrt 5))

```

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

## 3 Chapter 2: Building Abstractions with Data

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

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

### 3.1 2.1.1: Example: Arithmetic Operations for Rational Numbers

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

Note that the system proposed for representing and working with rational numbers has **abstraction barriers** isolating different parts of the system. The parts that use rational numbers don't know how the constructors and selectors for rational numbers work,

and the constructors and selectors use the underlying Lisp interpreter's pair functions without caring how they work.

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

## 3.2 Exercise 2.1

### 3.2.1 Text

```

1 (define (add-rat x y)
2   (make-rat (+ (* (number x) (denom y))
3     (* (number y) (denom x)))
4     (* (denom x) (denom y))))
5
6 (define (sub-rat x y)
7   (make-rat (- (* (number x) (denom y))
8     (* (number y) (denom x)))
9     (* (denom x) (denom y))))
10
11 (define (mul-rat x y)
12   (make-rat (* (number x) (number y))
13     (* (denom x) (denom y))))
14
15 (define (div-rat x y)
16   (make-rat (* (number x) (denom y))
17     (* (denom x) (number y))))
18
19 (define (equal-rat? x y)
20   (= (* (number x) (denom y))
21     (* (number y) (denom x))))

```

```

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

```

```

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

```

```

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

```

1/2  
1/6

### 3.2.2 Question

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

### 3.2.3 Answer

```

1 <<abs>>
2 (define (make-rat n d)
3   (cond ((not (or (< n 0)
4                 (< d 0)))
5         (cons n d))
6         ((and (< n 0)
7              (< d 0))
8          (cons (- n) (- d)))
9         (else
10          (cons (- (abs n)) (abs d)))))
11 (define (number x) (car x))
12 (define (denom x) (cdr x))
13
14 ;; Bonus: an attempt to optimize
15 (define (make-rat-opt n d)
16   (let ((nn (< n 0))
17         (dn (< d 0)))
18     (cond ((not (or nn dn))
19           (cons n d))
20           ((and nn dn)
21            (cons (- n) (- d)))
22           (else
23            (cons (- (abs n)) (abs d)))))

```

```

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

```

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

My “optimized” version shows no benefit at all:

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

## 3.3 Exercise 2.2

### 3.3.1 Question

Consider the problem of representing line segments in a plane. Each segment is represented as a pair of points: a starting point and an ending point. Define a constructor `make-segment` and selectors `start-segment` and `end-segment` that define the representation of segments in terms of points. Furthermore, a point can be represented as a pair of numbers: the  $x$  coordinate and the

$y$  coordinate. Accordingly, specify a constructor `make-point` and selectors `x-point` and `y-point` that define this representation. Finally, using your selectors and constructors, define a procedure `midpoint-segment` that takes a line segment as argument and returns its midpoint (the point whose coordinates are the average of the coordinates of the endpoints). To try your procedures, you'll need a way to print points:

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

### 3.3.2 Answer

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

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

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

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

And once again my bikeshedding is revealed:

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

## 3.4 Exercise 2.3

### 3.4.1 Question

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

design your system with suitable abstraction barriers, so that the same perimeter and area procedures will work using either representation?

### 3.4.2 Answer 1

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

```
1 (define (rect-area R)
2   (* (rect-height R)
3      (rect-width R)))
4
5 (define (rect-peri R)
6   (* 2
7      (+ (rect-height R)
8         (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  $D = A + (C - B) = C + (A - B)$ .

```
1 ;; AB = width
2 ;;(0,1) (1,1)
3 ;; A-----B
4 ;; |         | BC = height
5 ;; D-----C
6 ;;(0,0) (1,0)
7 ;; could be rotated any direction
8 <<square>>
9 <<make-point>>
10 (define (make-rect a b c)
11   (cons (cons a b) c))
12 (define (rect-a R)
13   (caar R))
14 (define (rect-b R)
15   (cdar R))
16 (define (rect-c R)
17   (cdr R))
```

```
18 ;(define (rect-d R)
19 ;  (make-point (x-point (rect-a R))
20 ;             (y-point (rect-c R))))
21 ;; Wait, this won't work if the rectangle is
   ↪ angled.
```

```
22
23 (define (sub-points a b)
24   (make-point (- (x-point a)
25                  (x-point b))
26               (- (y-point a)
27                  (y-point b))))
```

```
28
29 (define (add-points a b)
30   (make-point (+ (x-point a)
31                  (x-point b))
32               (+ (y-point a)
33                  (y-point b))))
```

```
34
35 (define (rect-d R)
36   (let ((a (rect-a R))
37         (b (rect-b R))
38         (c (rect-c R)))
39     (add-points a
40                (sub-points c b))))
41 (define (rect-d-alt R) ; should be
   ↪ mathematically identical.
42   (let ((a (rect-a R))
43         (b (rect-b R))
44         (c (rect-c R)))
45     (add-points c
46                (sub-points a b))))
```

```
47
48 ;; this is incorrect
49 ;(define (length-points a b)
50 ;  (let ((diffP (sub-points a b)))
51 ;    (+ (abs (x-point diffP))
52 ;       (abs (y-point diffP)))))
53 (define (length-points a b)
54   (let ((ax (x-point a))
55         (ay (y-point a))
56         (bx (x-point b))
57         (by (y-point b)))
58     (sqrt (+ (square (- ax bx))
59              (square (- ay by)))))
60
61 (define (rect-height R)
62   (abs (length-points (rect-b R)
63                       (rect-c R))))
```

```

64 (define (rect-width R)
65   (abs (length-points (rect-b R)
66                       (rect-a R))))
67
68 (define (length-segment seg)
69   (abs (length-points (start-segment seg)
70                       (end-segment seg))))

```

```

1  (load "mattcheck2.scm")
2
3  <<rect-4pt>>
4  <<rect-area-peri>>
5
6  (let* ((a (make-point 13 14))
7         (b (make-point 14 14))
8         (c (make-point 14 13))
9         (d (make-point 13 13))
10        (ABC (make-rect a b c))
11        (CDA (make-rect c d a))
12        (w (make-point -2.0 -2.0))
13        (x (make-point -0.5 -0.5))
14        (y (make-point -1.5 0.5))
15        (z (make-point -3.0 -1.0))
16        (WXY (make-rect w x y)))
17    (mattcheck "make-rect"
18              ABC
19              (cons (cons a b) c))
20    (mattcheck "rect-d and rect-d-alt (ABCD)"
21              (rect-d ABC)
22              (rect-d-alt ABC)
23              d)
24    (mattcheck "rect-d and rect-d-alt (CDAB)"
25              (rect-d CDA)
26              (rect-d-alt CDA)
27              b)
28    (mattcheck "rect-d and rect-d-alt (WXYZ)"
29              (rect-d WXY)
30              (rect-d-alt WXY)
31              z)
32    (mattcheck "rect-d and rect-d-alt (XYZW)"
33              (rect-d (make-rect x y z))
34              w)
35    (mattcheck "rect-height ABC"
36              (rect-height ABC)
37              1)
38    (mattcheck "rect-width ABC"
39              (rect-width ABC)
40              1)

```

```

41 (mattcheck "rect-height WXY"
42           (rect-height WXY)
43           1.4142135623730951)
44 (mattcheck "rect-width WXY"
45           (rect-width WXY)
46           2.1213203435596424)
47 (mattcheck "rect-area ABCD"
48           (rect-area ABC)
49           (rect-area CDA)
50           1)
51 (mattcheck "rect-area WXYZ"
52           (rect-area WXY)
53           3.0)
54 (mattcheck "rect-peri ABCD"
55           (rect-peri ABC)
56           4)
57 (mattcheck "rect-peri WXYZ"
58           (rect-peri WXY)
59           7.0710678118654755)

```

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

### 3.4.3 Answer 2

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

```

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

```

```

4   (cons (cons origin height)
5         (cons width angle)))
6
7   (define (rect-origin R)
8     (caar R))
9   (define rect-d rect-origin)
10  (define (rect-height R)
11    (cdar R))
12  (define (rect-width R)
13    (cadr R))
14  (define (rect-angle R)
15    (cddr R))
16
17  ;; I underestimated how much math this would
18  ↪ take.
19  (define (add-points a b)
20    (make-point (+ (x-point a)
21                  (x-point b))
22                (+ (y-point a)
23                  (y-point b))))
24
25  (define pi (* 4 (atan 1.0)))
26  (define (radian deg)
27    (* deg (/ pi 180.0)))
28  (define (vector-to-xy distance angle)
29    ;; rect-c: (cos(Theta),sin(Theta)) *
30    ↪ width
31    (make-point (* (cos (radian angle))
32                  distance)
33                (* (sin (radian angle))
34                  distance)))
35    ;; could also be rotated by 90 degrees
36    ↪ just by using
37    ;; (-sin(Theta),cos(Theta)) * height
38  (define (rect-c R)
39    (add-points
40      (rect-origin R)
41      (vector-to-xy (rect-width R) (rect-angle
42        ↪ R))))
43  (define (rect-a R)
44    (add-points
45      (rect-origin R)
46      (vector-to-xy (rect-height R)
47        (+ 90 (rect-angle R)))))

```

```

48  (vector-to-xy (rect-width R) (rect-angle
49    ↪ R))
50  (vector-to-xy (rect-height R)
51    (+ 90 (rect-angle R)))))

```

```

1   (load "mattcheck2.scm")
2
3   <<rect-ohwa>>
4   <<rect-area-peri>>
5
6   (let* ((a (make-point 13.0 14.0))
7          (b (make-point 14.0 14.0))
8          (c (make-point 14.0 13.0))
9          (d (make-point 13.0 13.0))
10         (ABC (make-rect d 1 1 0))
11         (CDA (make-rect b 1 1 180))
12         (w (make-point -2.0 -2.0))
13         (x (make-point -2.5 1.5))
14         (y (make-point -1.5 0.5))
15         (z (make-point -3.0 -1.0))
16         (wxy-height 1.4142135623730951)
17         (wxy-width 2.1213203435596424)
18         (WXY (make-rect z wxy-height wxy-width
19    ↪ 45)))
19     (mattcheck "make-rect"
20               ABC
21               (cons (cons d 1) (cons 1 0)))
22     (mattcheck "rect-b (ABCD)"
23               (rect-b ABC)
24               b)
25     (mattcheck "rect-b (CDAB)"
26               (rect-b CDA)
27               d)
28     (mattcheck "rect-b (WXYZ)"
29               (rect-b WXY)
30               x)
31     (mattcheck "rect-height"
32               (rect-height WXY)
33               wxy-height)
34     (mattcheck "rect-width"
35               (rect-width WXY)
36               wxy-width)
37     (mattcheck "rect-area ABCD"
38               (rect-area ABC)
39               (rect-area CDA)
40               1)
41     (mattcheck "rect-area WXYZ"
42               (rect-area WXY)

```

```

43      3.0)
44      (mattcheck "rect-peri ABCD"
45                (rect-peri ABC)
46                4)
47      (mattcheck "rect-peri WXYZ"
48                (rect-peri WXY)
49                7.0710678118654755))

```

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

### 3.5 2.1.3: What Is Meant by Data?

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

## 3.6 Exercise 2.4

### 3.6.1 Question

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

```

1  (define (cons x y)
2    (lambda (m) (m x y)))
3  (define (car z)
4    (z (lambda (p q) p)))

```

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

### 3.6.2 Answer

First, let's explain with the substitution model.

```

1  (cons 0 1)
2  (lambda (m) (m 0 1))
3
4  (car (lambda (m) (m 0 1)))
5  ((lambda (m) (m 0 1)) (lambda (p q) p))
6  (lambda (0 1) 0)
7  0
8  (cdr (lambda (m) (m 0 1)))
9  ((lambda (m) (m 0 1)) (lambda (p q) q))
10 (lambda (0 1) 1)
11 1

```

Now for implementation.

```

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

```

```

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

```

## 3.7 Exercise 2.5

optional

### 3.7.1 Question

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



### 3.7.2 Answer

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

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

First, let's make cons.

```
1 (define (cons-nnint a b)
2   (* (expt 2 a) (expt 3 b)))
3 (define (cons-nnint-debug a b) ;; DEBUG
4   (let* ((aa (expt 2 a))
5          (bb (expt 3 b))
6          (ab (* aa bb)))
7     (display aa)
8     (newline)
9     (display bb)
10    (newline)
11    (display ab)
12    (newline)
13    ab))
```

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

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

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

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

```
11                                     (cons-nnint (car
12   ↪ col) (cdr col)))
13                                     row)))
14   outputs)
15   inputs)))
```

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

Here are our scrambled numbers.

```
1 ;; To find a number of some base in some
2   ↪ column,
3 ;; First divide by unwantedbase for targetcol
4   ↪ number of times
5 <<repeated>>
6 (let ((targetcol 2)
7       (unwantedbase 3))
8   (map (lambda (row)
9         (map (lambda (item)
10                ((repeated (lambda (x)
11                              (/ x
12                                unwantedbase)) targetcol)
13                item))
14         row))
15   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.

```
1 <<logn>>
2 (let ((wantedbase 2))
3   (map (lambda (row)
```

```

4      (map (λ(item)
5            (format #f "~6,3f" (logn 2
6              ↪ item))))
7      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.

```

1  <<cons-nnint>>
2  <<logn>>
3  (use-srfis '(1))
4  (define (all-your-base ab unwanted wanted)
5    (if (equal? (modulo ab unwanted) 0)
6        (all-your-base (/ ab unwanted) unwanted
7          ↪ wanted)
8        (if (equal? (modulo ab wanted) 0)
9            (round (logn wanted ab))
10           "This number isn't a factor!")))
11 (define (car-nnint ab)
12   (all-your-base ab 3 2))
13 (define (cdr-nnint ab)
14   (all-your-base ab 2 3))
15 (let* ((initvalues '((2 3) (4 5) (7 2)))
16        (conslist (map (λ(x)
17                          (apply cons-nnint x))
18                        initvalues))
19        (carlist (map (λ(x)
20                        (car-nnint x))
21                       conslist))
22        (cdrlist (map (λ(x)
23                        (cdr-nnint x))
24                       conslist)))
25   (map (λ(x y) (cons x y))

```

```

26 (list "pairs" "cons'd" "car" "cdr")
27 (list initvalues conslist carlist
   ↪ cdrlist)))

```

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

## 3.8 Exercise 2.6

optional

### 3.8.1 Question

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

```

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

```

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

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

### 3.8.2 Answer

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

```

1 (define zero (λ (f) (λ (x) x)))
2 (define (add-1 n)
3   (λ (f) (λ (x)
4     (f ((n f) x)))))
5
6 (add-1 zero)
7 ((λ (f) (λ (x)
8   (f ((zero f) x)))))
9 ((λ (f) (λ (x)

```

```

10      (f ((λ (x) x) x) x))))
11  ((λ (f) (λ (x)
12      (f x))))

```

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

```

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

```

SUCCEED at 1 = 1+0

SUCCEED at 2 = 1+1+0

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

## 3.9 Exercise 2.7

### 3.9.1 Text

```

1  (define (add-interval x y)
2    (make-interval (+ (lower-bound x)
3        ↪ (lower-bound y))
4        (+ (upper-bound x)
5        ↪ (upper-bound y))))
6  (define (mul-interval x y)
7    (let ((p1 (* (lower-bound x) (lower-bound
8        ↪ y)))
9        (p2 (* (lower-bound x) (upper-bound
10        ↪ y)))
11        (p3 (* (upper-bound x) (lower-bound
12        ↪ y)))
13        (p4 (* (upper-bound x) (upper-bound
14        ↪ y))))
15      (make-interval (min p1 p2 p3 p4)
16          (max p1 p2 p3 p4))))

```

```

7      (p3 (* (upper-bound x) (lower-bound
8        ↪ y))))
9      (p4 (* (upper-bound x) (upper-bound
10        ↪ y))))
11      (make-interval (min p1 p2 p3 p4)
12          (max p1 p2 p3 p4))))
13
14  (define (div-interval x y)
15    (mul-interval
16      x
17      (make-interval (/ 1.0 (upper-bound y))
18          (/ 1.0 (lower-bound y)))))

```

### 3.9.2 Question

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

```

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

```

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

### 3.9.3 Answer

```

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

```

## 3.10 Exercise 2.8

### 3.10.1 Question

Using reasoning analogous to Alyssa's, describe how the difference of two intervals

may be computed. Define a corresponding subtraction procedure, called `sub-interval`.

### 3.10.2 Answer

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

```
1 (define (sub-interval x y)
2   (make-interval (- (lower-bound x)
3                     ↪ (upper-bound y))
4                   (- (upper-bound x)
5                     ↪ (lower-bound y))))
```

## 3.11 Exercise 2.9

### 3.11.1 Question

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

### 3.11.2 Answer

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

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

For this I wrote the following tests:

```
1 (load "mattcheck2.scm")
2 <<make-interval>>
3 <<sub-interval>>
4
5 (define (halve x)
6   (/ x 2))
7
8 (define (width-interval I)
9   (halve (- (upper-bound I)
10            (lower-bound I))))
11
12 (let* ((ia (make-interval 10.1 9.9))
13        (ib (make-interval 5.2 4.8))
14        (Aab (add-interval ia ib))
15        (Sab (sub-interval ia ib))
16        (Mab (mul-interval ia ib))
17        (Dab (div-interval ia ib)))
18   (mattcheck-float "ia width = roughly .1"
19                   0.1
20                   (width-interval ia))
21   (mattcheck-float "ib width = roughly .2"
22                   0.2
23                   (width-interval ib))
24   (mattcheck-float
25     ↪ "width addition is distributive"
26       (width-interval Aab)
27       (+ (width-interval ia)
28          (width-interval ib)))
29   (mattcheck-float
30     ↪ "width subtraction is distributive"
31       (width-interval Sab)
32       (- (width-interval ia)
33          (width-interval ib)))
34   (mattcheck-float
35     ↪ "width multiplication is distributive"
36       (width-interval Mab)
37       (* (width-interval ia)
38          (width-interval ib)))
39   (mattcheck-float
40     ↪ "width division is distributive"
41       (width-interval Dab)
42       (/ (width-interval ia)
43          (width-interval ib))))
```

<unknown-location>: warning: possibly unbound variable `mat  
 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: -0.100000000000000053
returned: 0.29999999999999998
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.

```

1 (load "mattcheck2.scm")
2 <<make-interval>>
3 <<sub-interval>>
4
5 (define (halve x)
6   (/ x 2))
7
8 (define (width-interval I)
9   (halve (- (upper-bound I)
10             (lower-bound I))))
11
12 (let* ((ia (make-interval 10.1 9.9))
13        (ib (make-interval 5.2 4.8))
14        (Aab (add-interval ia ib))
15        (Sab (sub-interval ia ib)))
16   (mattcheck-float "ia width = roughly .1"
17                     0.1
18                     (width-interval ia))
19   (mattcheck-float "ib width = roughly .2"
20                     0.2
21                     (width-interval ib))
22   (mattcheck-float
    ↪ "width(ia+ib) = width(ia) + width(ib)"

```

```

23 (width-interval Aab)
24 (+ (width-interval ia)
25     (width-interval ib)))
26 (mattcheck-float
    ↪ "width(ia-ib) = width(ia) + width(ib)"
27 (width-interval Sab)
28 (+ (width-interval ia)
29     (width-interval ib)))

```

SUCCEED at ia width = roughly .1

SUCCEED at ib width = roughly .2

SUCCEED at width(ia+ib) = width(ia) + width(ib)

SUCCEED at width(ia-ib) = width(ia) + width(ib)

## 3.12 Exercise 2.10

### 3.12.1 Question

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

### 3.12.2 Answer

```

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

```

## 3.13 Exercise 2.11

### 3.13.1 Question

In passing, Ben also cryptically comments: “By testing the signs of the endpoints of the intervals, it is possible to break `mul-interval` into nine cases, only one of which requires

more than two multiplications.” Rewrite this procedure using Ben’s suggestion.

### 3.13.2 Answer

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

Let’s examine the nine cases.

```

1 (use-modules (ice-9 format))
2 (use-srfis '(1))
3 (load "mattcheck2.scm")
4 <<make-interval>>
5 <<sub-interval>>
6
7 (define (matt-examine-mult f)
8   (let* ((pp (make-interval 3 2))
9          (pn (make-interval 3 -5))
10         (pn2 (make-interval 1 -0.5))
11         (nn (make-interval -5 -7))
12         (listofpairs (list
13                       (list pp pp)
14                       (list pp pn)
15                       (list pp nn)
16                       (list pn pp)
17                       (list pn pn)
18                       (list pn pn2) ;;<- edge
19                       (list pn nn) ;;
20                       (list nn pp)
21                       (list nn pn)
22                       (list nn nn))))
23     (givesign (λ(x)
24               (if (negative? x)
25                   "-"
26                   "+")))
27     (print-sign (λ(I)
28                  (format #f "~a ~a"
29                          (givesign
30                            (upper-bound I))
31                          (givesign
32                            (lower-bound I))))))
31     (print-int (λ(I)
32                  (format #f "~a/~a"
33                          (upper-bound I)
34                          (lower-bound
35                            I)))))
35     (print-ints (λ(I J)

```

```

36         (format #f "~a times ~a"
37                 (print-int I)
38                 (print-int J))))
39     (results (map (λ(p)
40                   (apply f p))
41                 listofpairs)))
42   (list
43     (map (λ(p)
44           (apply print-ints p))
45          listofpairs)
46     (map print-int results)
47     (map (λ(I)
48           (print-sign I))
49          results)
50     (map (λ(p)
51           (format #f "~a // ~a"
52                 (print-sign (car p))
53                 (print-sign (cadr p))))
54          listofpairs))))
55   (cons
56     (list "problem" "result" "signs"
57           "problem signs")
57     (apply zip
58       (matt-examine-mult mul-interval)))

```

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

```

1 (define (mul-interval-opt x y)
2   (let ((xu (upper-bound x))
3         (xl (lower-bound x))
4         (yu (upper-bound y))
5         (yl (lower-bound y)))
6     (define p? positive?)
7     (define n? negative?)
8     (define (check-signs? a b x y) ;; pass
9       functions
10      (and (a xu)

```

```

10      (b xl)
11      (x yu)
12      (y yl)))
13 (define (same-signs?)
14   (or (check-signs? p? p? p? p?)
15       (check-signs? n? n? n? n?)))
16 (define (alt-signs?)
17   (or (check-signs? p? p? n? n?)
18       (check-signs? n? n? p? p?)))
19 (cond ((same-signs?)
20       (make-interval (* xu yu)
21                       (* xl yl)))
22      ((alt-signs?)
23       (make-interval (* xl yu)
24                       (* xu yl)))
25      ((check-signs? p? p? p? n?)
26       (make-interval (* xu yu)
27                       (* xu yl)))
28      ((check-signs? p? n? p? p?)
29       (make-interval (* xu yu)
30                       (* xl yu)))
31      ((check-signs? p? n? p? n?)
32       (let ((p1 (* xu yu))
33             (p2 (* xu yl))
34             (p3 (* xl yu))
35             (p4 (* xl yl)))
36         (make-interval (max p1 p2 p3 p4)
37                         (min p1 p2 p3
38                               ↪ p4))))
38      ((check-signs? p? n? n? n?)
39       (make-interval (* xl yl)
40                       (* xu yl)))
41      ((check-signs? n? n? p? n?)
42       (make-interval (* xl yl)
43                       (* xl yu))))))

```

```

1 (use-modules (ice-9 format))
2 (use-srfis '(1))
3 (load "mattcheck2.scm")
4 <<make-interval>>
5 <<sub-interval>>
6 <<mul-interval-opt>>
7
8 (define (matt-mult-consistency f1 f2)
9   (let* ((pp (make-interval 3 2))
10          (pn (make-interval 3 -5))
11          (pn2 (make-interval 1 -0.5))
12          (nn (make-interval -5 -7))

```

```

13      (listofpairs (list
14                    (list "pp*nn" pp pp)
15                    (list "pp*pn" pp pn)
16                    (list "pp*nn" pp nn)
17                    (list "pn*pp" pn pp)
18                    (list "pn*pn" pn pn)
19                    (list "pn*pn2" pn pn2)
20                    ↪ ;;<- edge case to catch incomplete
21                    (list "pn*nn" pn nn)
22                    ↪ ;; multiplication functions
23                    (list "nn*pp" nn pp)
24                    (list "nn*pn" nn pn)
25                    (list "nn*nn" nn nn))))
26      (map (λ(l)
27            (mattcheck (car l)
28                        (apply f1 (cdr l))
29                        (apply f2 (cdr l))))
30            listofpairs)))
31 (matt-mult-consistency mul-interval
32 ↪ mul-interval-opt)

```

SUCCEED at pp\*nn  
 SUCCEED at pp\*pn  
 SUCCEED at pp\*nn  
 SUCCEED at pn\*pp  
 SUCCEED at pn\*pn  
 SUCCEED at pn\*pn2  
 SUCCEED at pn\*nn  
 SUCCEED at nn\*pp  
 SUCCEED at nn\*pn  
 SUCCEED at nn\*nn

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

So as expected, about twice as fast!

## 3.14 Exercise 2.12

### 3.14.1 Question

After debugging her program, Alyssa shows it to a potential user, who complains that her program solves the wrong problem. He wants a program that can deal with numbers represented as a center value and an additive tolerance; for example, he wants to work with intervals such as  $3.5 \pm 0.15$  rather than  $[3.35,$

3.65]. Alyssa returns to her desk and fixes this problem by supplying an alternate constructor and alternate selectors:

```
1 (define (make-center-width c w)
2   (make-interval (- c w) (+ c w)))
3 (define (center i)
4   (/ (+ (lower-bound i) (upper-bound i)) 2))
5 (define (width i)
6   (/ (- (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

```
1 (define (reciprocal x)
2   (/ 1 x))
```

```
1 <<reciprocal>>
2 <<make-interval>>
3 <<sub-interval>>
4 <<interval-center-width>>
5
6 (define (make-center-percent c pt)
7   (let ((pp (* c
8             (* pt 0.01))))
9     (make-interval (- c pp) (+ c pp))))
10 (define (percent I)
11   (* 100.0
12     (/ (width I)
13        (center I))))
```

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

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

## 3.15 Exercise 2.13 optional

### 3.15.1 Question

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



### 3.15.2 Answer

I should've written this function a while ago.

```

1 (use-modules (ice-9 format))
2 (define (stringit . args)
3   (string-append
4     (apply string-append
5       (map (λ(x)
6             (format #f "~a " x))
7             args))))
8 (define (echo . args)
9   (format #t "~&a~%" (apply stringit args)))

```

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

```

1 <<echo>>
2 <<interval-percent>>
3 <<mul-interval-opt>>
4
5 (let* ((i1 (make-center-width 100 5))
6        (i2 (make-center-width 200 5))
7        (M12 (mul-interval-opt i1 i2)))
8   (echo "intervals 1 and 2:" i1 i2)
9   (echo "width of 1 and 2:" (width i1) (width
10    ↪ i2))
11   (echo "percent of 1 and 2:" (percent i1)
12    ↪ (percent i2))
13   (echo "i1*i2 = " M12)
14   (echo "width M12:" (width M12))
15   (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  $\text{percent}(A \times B) = \text{percent}(A) + \text{percent}(B)$ ?

```

1 <<echo>>
2 <<interval-percent>>
3 <<mul-interval-opt>>
4
5 (let* ((i1 (make-center-percent 40 0.1))
6        (i2 (make-center-percent 200 0.4))
7        (M12 (mul-interval-opt i1 i2)))

```

```

8   (echo "percent of 1 and 2:" (percent i1)
9    ↪ (percent i2))
10   (echo "percent M12:" (percent M12)))

```

percent of 1 and 2: 0.099999999999999788 0.400000000000000563  
percent M12: 0.499998000000

## 3.16 Exercise 2.14

### 3.16.1 Question

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

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

and

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

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

```

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

```

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

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

### 3.16.2 Answer

```

1  <<echo>>
2  <<interval-percent>>
3  <<mul-interval-opt>>
4  <<par-resistors>>
5
6  (let* ((A (make-center-percent 10 1))
7         (B (make-center-percent 10 0.01))
8         (p1 (par1 A B))
9         (p2 (par2 A B)))
10    (echo "A,B:" A B)
11    (echo "par1(A,B):" p1)
12    (echo "par2(A,B):" p2)
13    (echo "percent(par1):" (percent p1))
14    (echo "percent(par2):" (percent p2))
15    (echo "center(par1):" (center p1))
16    (echo "center(par2):" (center p2)))
17
18  (echo |
19  ↪ "So these two have inconsistent effects on the width."
20  ↪ )
21  (newline)
22  (echo |
23  ↪ "It should also be noted that floating-point errors accumulate."
24  ↪ )
25  (echo |
26  ↪ "Take a look at the error on these (correct answer is 1)."
27  ↪ )
28
29  (let* ((A (make-center-percent 10 1))
30         (p1 (div-interval
31              (div-interval
32                (mul-interval A A)
33                A)
34                A)
35              (p2 (div-interval
36                   (div-interval
37                     (mul-interval A A)
38                     A)
39                     A))))
40    (echo "p1:" (center p1))
41    (echo "p2:" (center p2)))

```

```

31      (div-interval
32        (mul-interval
33          (mul-interval A A)
34          A)
35        A)
36      (p2 (div-interval
37           (div-interval
38             (mul-interval A A)
39             A)
40           A))))
41    (echo "p1:" (center p1))
42    (echo "p2:" (center p2)))

```

```

A,B: (10.1 . 9.9) (10.001 . 9.999)
par1(A,B): (5.076139504497713 . 4.924635590269141)
par2(A,B): (5.025128103079449 . 4.974626865671642)
percent(par1): 1.5149217214958663
percent(par2): 0.5050247487625606
center(par1): 5.000387547383427
center(par2): 4.999877484375546

```

So these two have inconsistent effects on the width.

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

```

p1: 1.0008001600240033
p2: 1.0018006601460259

```

## 3.17 Exercise 2.15

### 3.17.1 Question

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

### 3.17.2 Answer

If I am correct in understanding that "uncertain number" means "a number with an error tolerance", then `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 exam-

ple, take these expressions:

$$A + A = 2A$$

$$A - A = 0$$

$$A/A = 1$$

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

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

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

### 3.18 Exercise 2.16

optional

#### 3.18.1 Question

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

#### 3.18.2 Answer

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

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

## 3.19 2.2: Hierarchical Data and the Closure Property

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

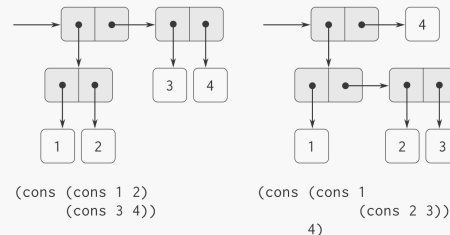


Figure 2.3: Two ways to combine 1, 2, 3, and 4 using pairs.

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

#### Definitions of “closure”

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

### 3.20 2.2.1: Representing Sequences

**sequence** An ordered collection of data objects.

**list** A sequence of cons pairs.

```

1 (cons 1
2     (cons 2
3         (cons 3
4             (cons 4 nil))))
5 (list 1 2 3 4)
6 ;; 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`:

```

1 (define (map f l)
2   (if (null? l)
3       #nil
4       (cons (f (car l))
5             (map f (cdr l)))))
6
7 (map (lambda (x) (* x 2))
8      (list 1 2 3 4))

```

| 2 | 4 | 6 | 8 |

## 3.21 Exercise 2.17

### 3.21.1 Question

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

```

1 (last-pair (list 23 72 149 34))
2 ;; (34)

```

### 3.21.2 Answer

```

1 (define (last-pair l)
2   (let ((a (car l))
3         (d (cdr l)))
4     (if (= 1 (length d))
5         d
6         (last-pair d))))

```

```

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

```

| 34 |

## 3.22 Exercise 2.18

### 3.22.1 Question

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

```

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

```

### 3.22.2 Answer

```

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

```

```

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

```

34 149 72 23

## 3.23 Exercise 2.19

### 3.23.1 Question

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

the procedure `count-change` (which knows that there are five kinds of U.S. coins). It would be nicer to be able to supply a list of coins to be used for making change.

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

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

We could then call `cc` as follows:

```
1 (cc 100 us-coins)
2 ; 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:

```
1 (define (cc amount coin-values)
2   (cond ((= amount 0) 1)
3         ((or (< amount 0) (no-more?
4   ↪ coin-values)) 0)
5         (else
6          (+ (cc amount
7                (except-first-denomination
8                  coin-values))
9            (cc (- amount
10                  (first-denomination
11                  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

```
1 <<cc-lists>>
2
3 (define us-coins
4   (list 50 25 10 5 1))
5 (define uk-coins
6   (list 100 50 20 10 5 2 1 0.5))
7
8 (define first-denomination car)
9 (define except-first-denomination cdr)
10 (define no-more? null?)
```

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

| 292 | 292 | 292 |

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

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

## 3.24 Exercise 2.20

### 3.24.1 Question

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

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

### 3.24.2 Answer

```

1 (define (same-parity . rest)
2   (define same?
3     (if (even? (car rest))
4         even?
5         odd?))
6   (define (iter l results)
7     (if (null? l)
8         results
9         (let ((a (car l)))
10          (iter (cdr l)
11                (if (same? a)
12                    (cons a results)
13                    results)))))
14   (iter (reverse rest) '()))
15
16 ;; Attempting to remove the reversing
17 (define (same-parity2 . args)
18   (define first (car args))
19   (define same?
20     (if (even? first)
21         even?
22         odd?))
23   (define (iter l results)
24     (if (null? l)
25         results
26         (let ((a (car l))
27               (d (cdr l)))
28           (if (same? a)
29               (iter d (append results
30                               (cons a #nil)))
31               (iter d results)))))
32   (iter (cdr args) (cons first #nil)))

```

```

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

```

```

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

```

```

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

```

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

## 3.25 Exercise 2.21

### 3.25.1 Question

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

```

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

```

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

```

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

```

### 3.25.2 Answer

```

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

```

```

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

```

```

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

```

## 3.26 Exercise 2.22

### 3.26.1 Questions

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

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

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

This doesn't work either. Explain.

### 3.26.2 Answer

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

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

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

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

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

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

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

## 3.27 Exercise 2.23

### 3.27.1 Question

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

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

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

### 3.27.2 Answer

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

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

all your base are belong to us

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

textbook version: (1 (2 (3 4)))  
cons'd version: (1 (2 (3 4)))

Dot and box version:

## 3.28 Exercise 2.24

### 3.28.1 Text Definitions

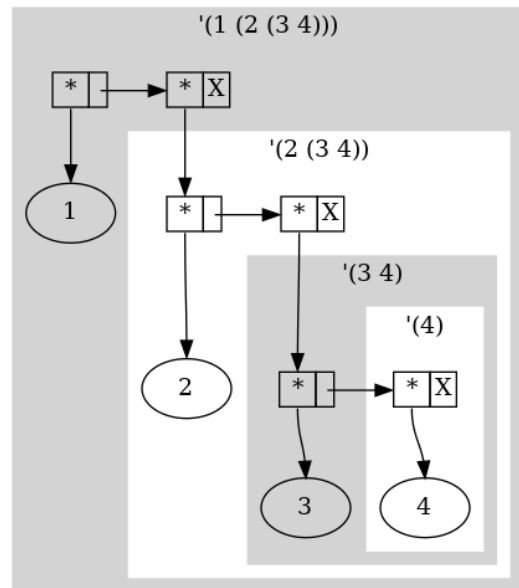
```
1 (define (count-leaves x)
2   (cond ((null? x) 0)
3         ((not (pair? x)) 1)
4         (else (+ (count-leaves (car x))
5                   (count-leaves (cdr x))))))
```

### 3.28.2 Question

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

### 3.28.3 Answer

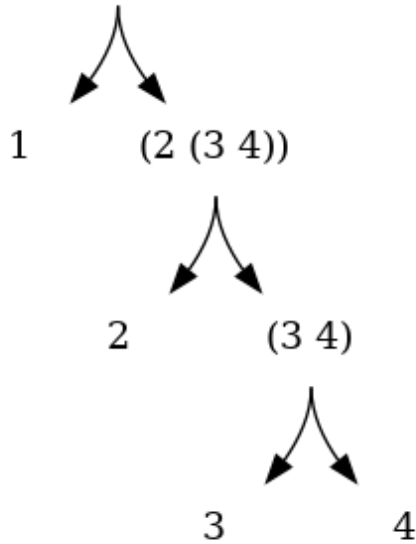
This is sort of a trick question – on first reading, I read it like a series of `cons` statements. Looking again, though, I can see that the correct formulation is as follows:



Tree version:



(1 (2 (3 4)))



### 3.29 Exercise 2.25

#### 3.29.1 Question

Give combinations of `cars` and `cdrs` that will pick 7 from each of the following lists:

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

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

7 7 7

### 3.30 Exercise 2.26

#### 3.30.1 Question

Suppose we define `x` and `y` to be two lists:

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

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

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

#### 3.30.2 Answer

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

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

### 3.31 Exercise 2.27

#### 3.31.1 Question

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

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

```

5 ;; ((3 4) (1 2))
6 (deep-reverse x)
7 ;; ((4 3) (2 1))

```

### 3.31.2 Answer

```

1 (define (deep-reverse l)
2   (define len (length l))
3   (define (iter i result)
4     (if (< (1- len) i)
5         result
6         (iter (+ i 1)
7               (cons (let ((here (list-ref l
8                               ↪ i)))
9                       (if (pair? here)
10                           (deep-reverse here)
11                           here)))
12                     result))))
13 (iter 0 '())

```

```

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

```

```

1 ((4 3) (2 1))

```

## 3.32 Exercise 2.28

### 3.32.1 Question

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

```

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

```

### 3.32.2 Answer

```

1 (define (fringe l)
2   (if (null? l)
3       #nil
4       (let ((a (car l))

```

```

5         (d (cdr l)))
6     (append (if (pair? a)
7                 (fringe a)
8                 (list a))
9             (fringe d))))

```

```

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

```

```
(1 2 3 4 5)
```

## 3.33 Exercise 2.29: Binary Mobiles

### 3.33.1 Text Definitions

```

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

```

### 3.33.2 Question A: Selectors

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

### 3.33.3 Answer A

```

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

```

### 3.33.4 Question B: total-weight

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

### 3.33.5 Answer B

```

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

```

```

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

```

| 10 | 8 |

### 3.33.6 Question C: Balancing

A mobile is said to be **balanced** if the torque applied by its top-left branch is equal to that applied by its top-right branch (that is, if the length of the left rod multiplied by the weight hanging from that rod is equal to the corresponding product for the right side) and if each of the submobiles hanging off its branches is balanced. Design a predicate that tests whether a binary mobile is balanced.

### 3.33.7 Answer C

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

```

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

```

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

```

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

```

```

8      (total-weight leftS))
9      (if (number? rightS)
10         rightS
11         (total-weight rightS))))))

```

```

1  <<echo>>
2  <<mobile-selectors-list>>
3  <<mobile-total-weight-2>>
4  <<mobile-balanced-dumb>>
5  (define M1 ;; all segments unbalanced
6    (make-mobile
7      (make-branch 5 5) ;; torque 25 = 5*5
8      (make-branch 1 ;; torque 5 = (2 + 3) * 1
9                (make-mobile (make-branch 2 2)
10                           (make-branch 2
11                                     3))))))
12  (define M2 ;; all segments balanced as they
13    are duplicates
14    (make-mobile
15      (make-branch 2 ;; torque 8 = 2*(2+2)
16                (make-mobile
17                  (make-branch 2 2)
18                  (make-branch 2 2)))
19      (make-branch 2
20                (make-mobile
21                  (make-branch 2 2)
22                  (make-branch 2 2))))))
23  (define M3 ;; equal torque, but one segment is
24    unbalanced.
25    (make-mobile
26      (make-branch 2 ;; torque 12 = 2*(4+2)
27                (make-mobile
28                  (make-branch 2 4) ;; torque 8
29                  (make-branch 4 2))) ;torque 8
30      (make-branch 3 ;; torque 12 = 3*(1+3)
31                (make-mobile
32                  (make-branch 1 1) ;; torque 1
33                  (make-branch 1 3)))))) ;;
34  (define M4 ;; equal torque
35    (make-mobile
36      (make-branch 2 ;; torque 12 = 2*(4+2)
37                (make-mobile
38                  (make-branch 2 4) ;; torque 8
39                  (make-branch 4 2))) ;torque 8
40      (make-branch 3 ;; torque 12 = 3*(1+3)
41                (make-mobile
42                  (make-branch 1 1) ;; torque 1
43                  (make-branch 1 3)))))) ;;

```

```

36      (make-branch 4 2))) ;torque 8
37      (make-branch 3 ;; torque 12 = 3*(1+3)
38                (make-mobile
39                  (make-branch 3 1) ;; torque 3
40                  (make-branch 1 3)))))) ;;
41  (define (isbalanced? Name Status)
42    (define Success "is balanced!")
43    (define Failure "is not balanced!")
44    (if (eq? Status #t)
45        (echo Name Success)
46        (echo Name Failure)))
47  (isbalanced? "M1" (balanced? M1))
48  (isbalanced? "M2" (balanced? M2))
49  (isbalanced? "M3" (balanced? M3))
50  (isbalanced? "M4" (balanced? M4))

```

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

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

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

### 3.33.8 Question D: Implementation shakeup

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

```

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

```

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

### 3.33.9 Answer D

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

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

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

```
1 <<echo>>
2 <<mobile-selectors-cons>>
3 <<mobile-total-weight-2>>
4 <<mobile-balanced-dumb>>
5 (define M1 ;; all segments unbalanced
6   (make-mobile
7     (make-branch 5 5) ;; torque 25 = 5*5
8     (make-branch 1 ;; torque 5 = (2 + 3) * 1
9       (make-mobile (make-branch 2 2)
10                    (make-branch 2
11                      3))))))
12 (define M2 ;; all segments balanced as they
13   are duplicates
14   (make-mobile
15     (make-branch 2 ;; torque 8 = 2*(2+2)
16       (make-mobile
17         (make-branch 2 2)
18         (make-branch 2 2)))
19     (make-branch 2
20       (make-mobile
21         (make-branch 2 2)
22         (make-branch 2 2))))))
23 (define M3 ;; equal torque, but one segment is
24   unbalanced.
25   (make-mobile
26     (make-branch 2 ;; torque 12 = 2*(4+2)
27       (make-mobile
28         (make-branch 2 4) ;; torque 8
29         (make-branch 4 2))) ;torque 8
30     = 2*4
31     = 4*2
```

```
27 (make-branch 3 ;; torque 12 = 3*(1+3)
28   (make-mobile
29     (make-branch 1 1) ;; torque 1
30     (make-branch 1 3)))))) ;;
31 ↪ torque 3
32 (define M4 ;; equal torque
33   (make-mobile
34     (make-branch 2 ;; torque 12 = 2*(4+2)
35       (make-mobile
36         (make-branch 2 4) ;; torque 8
37         (make-branch 4 2))) ;torque 8
38     = 2*4
39     = 4*2
40     (make-branch 3 ;; torque 12 = 3*(1+3)
41       (make-mobile
42         (make-branch 3 1) ;; torque 3
43         (make-branch 1 3)))))) ;;
44 ↪ torque 3
45 (define (isbalanced? Name Status)
46   (define Success "is balanced!")
47   (define Failure "is not balanced!")
48   (if (eq? Status #t)
49       (echo Name Success)
50       (echo Name Failure)))
51 (isbalanced? "M1" (balanced? M1))
52 (isbalanced? "M2" (balanced? M2))
53 (isbalanced? "M3" (balanced? M3))
54 (isbalanced? "M4" (balanced? M4))
```

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

### 3.34 Exercise 2.30

#### 3.34.1 Question

Define a procedure `square-tree` analogous to the `square-map` procedure of 3.25. That is, `square-tree` should behave as follows:

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

Define `square-tree` both directly (i.e., without using

any higher-order procedures) and also by using `map` and recursion.

### 3.34.2 Answer

```

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

```

```

1 (load "mattcheck2.scm")
2 <<square-tree>>
3 (let ((testlist
4       (list 1
5             (list 2 (list 3 4) 5)
6             (list 6 7)))
7       (answer
8         (list 1
9               (list 4 (list 9 16) 25)
10              (list 36 49))))
11   (mattcheck "square-tree-discrete"
12             (square-tree-discrete testlist)
13             answer)
14   (mattcheck "square-tree-map"
15             (square-tree-map testlist)
16             answer))

```

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

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

```

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

```

```

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

```

## 3.35 Exercise 2.31

### 3.35.1 Question

Abstract your answer to 3.34 to produce a procedure `tree-map` with the property that `square-tree` could be defined as

```

1 (define (square-tree tree) (tree-map square
2   ↪ tree))

```

### 3.35.2 Answer

```

1 (define (tree-map f tree)
2   (cond ((null? tree) '())
3         ((not (pair? tree)) (f tree))
4         (else (cons (tree-map f (car tree))
5   ↪ (tree-map f (cdr tree))))))

```

```

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

```

SUCCEED at square-tree-tm

### 3.36 Exercise 2.32

#### 3.36.1 Question

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

```

1 (define (subsets s)
2   (if (null? s)
3       (list nil)
4       (let ((rest (subsets (cdr s))))
5         (append rest (map <??> rest)))))

```

#### 3.36.2 Answer

```

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

```

```
rest))))))
```

```

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

```

SUCCEED at subsets

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

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

```

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

```

```

1 <<subsets>>
2 <<echo-return>>
3 (define (subsets-debug s)
4   (echo "Enter with" s "{")
5   (let ((a1
6         (if (null? s)
7             (echo-return (list '()))
8             (let ((rest (subsets-debug (cdr
9   ↪ s))))
10      (echo "[ iter" (car s) "over"
11      ↪ rest "]" )
12      (append rest (map (lambda (x)

```

```

11                                     (let ((y
12 ↪ (cons (car s) x)))
13                                     (format
14 ↪ #t "~a " y)
15                                     y))
16                                     rest))))))
17 (let ((answer
18       (list '()
19             (list 3)
20             (list 2) (list 2 3)
21             (list 1) (list 1 3)
22             (list 1 2) (list 1 2 3))))
23   (subsets-debug (list 1 2 3))
24   answer)

```

```

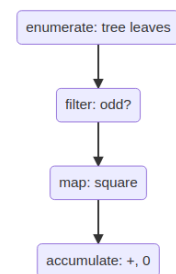
Enter with (1 2 3) {
Enter with (2 3) {
Enter with (3) {
Enter with () {
  (())
} end ()
[ iter 3 over (()) ]
(3)
} end (3)
[ iter 2 over (() (3) (2) (2 3)) ]
(2) (2 3)
} end (2 3)
[ iter 1 over (() (3) (2) (2 3)) ]
(1) (1 3) (1 2) (1 2 3)
} end (1 2 3)

```

### 3.37 2.2.3: Sequences as Conventional Interfaces

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

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



### 3.38 Exercise 2.33: The flexibility of (accumulate)

#### 3.38.1 Text Definitions

```

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

```

#### 3.38.2 Question

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

```

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

```

#### 3.38.3 Answer

```

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

```



```

8 (accumulate (lambda (x y)
9             (1+ y))
10            0 sequence))

```

```

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

```

SUCCEED at map  
 SUCCEED at append  
 SUCCEED at length

### 3.39 Exercise 2.34

#### 3.39.1 Question

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

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

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

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

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

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

```

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

```

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

```

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

```

#### 3.39.2 Answer

```

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

```

```

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

```

SUCCEED at horner-eval

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

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

Into this operation, left to right:

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

### 3.40 Exercise 2.35

#### 3.40.1 Question

Redefine `count-leaves` from 2.2.2 as an accumulation:

#### 3.40.2 Answer

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

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

SUCCEED at count-leaves-acc

### 3.41 Exercise 2.36: Accumulate across multiple lists

#### 3.41.1 Question

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

should be the sequence `(22 26 30)`. Fill in the missing expressions in the following definition of `accumulate-n`:

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

#### 3.41.2 Answers

```
1 (define (accumulate-n op init seqs)
2   (if (null? (car seqs))
3       '()
4       (cons (accumulate op init
5                         (map car seqs))
6             (accumulate-n op init
7                           (map cdr seqs)))))
```

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

SUCCEED at accumulate-n

### 3.42 Exercise 2.37: Enter the matrices

#### 3.42.1 Question

See full quote in book.

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

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

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

Define these operations:

```

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

```

### 3.42.2 Answer

```

1 (define (dot-product v w)
2   (accumulate + 0 (map * v w)))
3 (define (matrix-*-vector m v)
4   (map (λ(row)
5         (dot-product v row))
6        m))
7 (define (transpose mat)
8   (accumulate-n cons '() mat))
9 (define (matrix-*-matrix m n)
10  (let ((cols (transpose n)))
11    (map (λ(row)
12          (map (λ(col)
13                (dot-product row col))
14               cols))
15         m)))

```

```

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

```

```

20 70)
21 (mattcheck "matrix-*-vector"
22           (matrix-*-vector m1 v1)
23           (list 30 70))
24 (mattcheck "matrix-*-matrix"
25           (matrix-*-matrix m1 m1t)
26           m1-*-m1t))

```

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

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

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

```

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

```

But much harder to comprehend Lisp code like this:

```

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

```

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

## 3.43 Exercise 2.38: fold-right

### 3.43.1 Question A

The `accumulate` procedure is also known as `fold-right`, because it combines the first element of the sequence with the result of combining all the elements to the right. There

is also a `fold-left`, which is similar to `fold-right`, except that it combines elements working in the opposite direction:

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

What are the values of

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

### 3.43.2 Answer A

```
1 <<accumulate>>
2 (define fold-right accumulate)
3 <<fold-left>>
4 <<echo>>
5 (echo "(fold-right / 1 (list 1 2 3)):"
6   ↪ (fold-right / 1 (list 1 2 3)))
6 (echo "(fold-left / 1 (list 1 2 3)):"
7   ↪ (fold-left / 1 (list 1 2 3)))
7 (echo "(fold-right list nil (list 1 2 3)):"
8   ↪ (fold-right list '() (list 1 2 3)))
8 (echo "(fold-left list nil (list 1 2 3)):"
9   ↪ (fold-left list '() (list 1 2 3)))
```

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

### 3.43.3 Question B

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

### 3.43.4 Answer B

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

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

SUCCEED at commutative addition  
SUCCEED at commutative multiplication

## 3.44 Exercise 2.39: reverse via fold

### 3.44.1 Question

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

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

### 3.44.2 Answer

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

```

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

```

Now to the problem.

```

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

```

```

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

```

SUCCEED at reverse-fr

SUCCEED at reverse-fl

## 3.45 Exercise 2.40: unique-pairs

### 3.45.1 Text Definitions

```

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

```

```

1 <<fold-compat>>
2 (define (flatmap proc seq)

```

```

3   (accumulate append '() (map proc seq)))

```

```

1 <<flatmap>>
2 <<enumerate-interval>>
3 <<prime-smallest-divisor>>
4 (define (prime-sum? pair)
5   (prime? (+ (car pair) (cadr pair))))
6 (define (make-pair-sum pair)
7   (list (car pair) (cadr pair) (+ (car pair)
8   ↪ (cadr pair))))
9 (define (prime-sum-pairs n)
10  (map make-pair-sum
11       (filter prime-sum? (flatmap
12                             (lambda (i)
13                               (map (lambda (j)
14   ↪ (list i j))
15   ↪ (enumerate-interval 1 (- i 1))))
16   ↪ (enumerate-interval
17   ↪ 1 n)))))

```

### 3.45.2 Question

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

### 3.45.3 Answer

```

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

```

```

1 (load "mattcheck2.scm")
2 <<unique-pairs>>
3 (let ((target 5))
4   (answer (list (list 2 1)(list 3 1)(list
5   ↪ 3 2)(list 4 1)

```

```

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

```

SUCCEED at unique-pairs

SUCCEED at prime-sum-pairs with unique-pairs

### 3.46 Exercise 2.41: Ordered triples of positive integers

#### 3.46.1 Question

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

#### 3.46.2 Answer

```

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

```

```

1 (define (triplets-sum n s)
2   (filter (λ(triplet)
3           (= s (fold + 0 triplet))))
4   (unique-triplets n)))

```

```

1 (load "mattcheck2.scm")
2 <<unique-triplets>>
3 <<triplets-sum>>
4 (let ((answer-a
5       (list (list 3 2 1) (list 4 2 1)
6             (list 4 3 1) (list 4 3 2)
7             (list 5 2 1) (list 5 3 1)
8             (list 5 3 2) (list 5 4 1)
9             (list 5 4 2) (list 5 4 3))))
10   (answer-b
11   (list (list 5 3 2) (list 5 4 1)
12         (list 6 3 1) (list 7 2 1))))
13 (mattcheck "unique-triplets"
14   (unique-triplets 5)
15   answer-a)
16 (mattcheck "triplets-sum"
17   (triplets-sum 7 10)
18   answer-b))

```

SUCCEED at unique-triplets

SUCCEED at triplets-sum

### 3.47 Exercise 2.42: Eight Queens

#### 3.47.1 Question

The “eight-queens puzzle” asks how to place eight queens on a chessboard so that no queen is in check from any other.

```

1 <<flatmap>>
2 <<enumerate-interval>>
3 (define (queens board-size)
4   (define (queen-cols k)
5     (if (= k 0)
6         (list empty-board)
7         (filter
8             (lambda (positions) (safe? k
↪ positions))
9             (flatmap
10              (lambda (rest-of-queens)
11                (map (lambda (new-row)

```

```

12         (adjoin-position
13           new-row k rest-of-queens))
14     (enumerate-interval 1
15       board-size)))
16     (queen-cols (- k 1))))))
17   (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  $k$

```

20     (compare
21       (older-positions board)))
22     (and-map (lambda (pos)
23               (not (or
24                     (same-row? new pos)
25                     (diagonal?
26                       new pos))))
27             compare)))

```

```

1  <<queens-txt>>
2  <<queens-mine>>
3  (load "mattcheck2.scm")
4  (let ((q4 '(((3 4) (1 3) (4 2)
5              (2 1))
6              ((2 4) (4 3) (1 2)
7              (3 1)))))
9    (q11l 2680))
10   (mattcheck "queens"
11             (list (queens 4)
12                   (length
13                     (queens 11)))
14             (list q4 q11l)))

```

SUCCEED at queens

### 3.47.2 Answer

```

1  (define empty-board '())
2  (define (adjoin-position new-row
3    column rest-of-queens)
4    (cons (list new-row column)
5          rest-of-queens))
6  (define (newer-position board)
7    (car board))
8  (define (older-positions board)
9    (cdr board))
10 (define (get-row position)
11   (car position))
12 (define (get-column position)
13   (cadr position))
14 (define (safe? k board)
15   (define (same-row? a b)
16     (= (get-row a) (get-row b)))
17   (define (diagonal? a b)
18     (let ((row-diff (abs (-
19       (get-row a) (get-row b))))
20           (col-diff (abs (-
21       (get-column a) (get-column
22       b)))))
23       (= row-diff col-diff)))
24   (let* ((new (newer-position
25     board))

```

## 3.48 Exercise 2.43: Louis' queens

### 3.48.1 Question

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  $6 \times 6$  case.) When Louis asks Eva Lu Ator for help, she points out that he has interchanged the order of the nested mappings in the `flat-map`, writing it as

```

1  (flatmap
2    (lambda (new-row)
3      (map (lambda (rest-of-queens)

```

```

4      (adjoin-position
↪ new-row k rest-of-queens))
5      (queen-cols (- k 1)))
6      (enumerate-interval 1
↪ board-size))

```

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

### 3.48.2 Answer

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

```

1  <<queens-txt>>
2  <<queens-mine>>
3  (load "../mattbench.scm")
4
5  (define (queens-louis board-size)
6    (define (queen-cols k)
7      (if (= k 0)
8          (list empty-board)
9          (filter
10             (lambda (positions)
↪ (safe? k positions))
11             (flatmap
12                (lambda (new-row)
13                  (map (lambda
↪ (rest-of-queens)
14
15                  (adjoin-position new-row k
↪ rest-of-queens))
16                  (queen-cols (-
↪ k 1))))))
17    (enumerate-interval 1
↪ board-size))))
18    (queen-cols board-size))

```

```

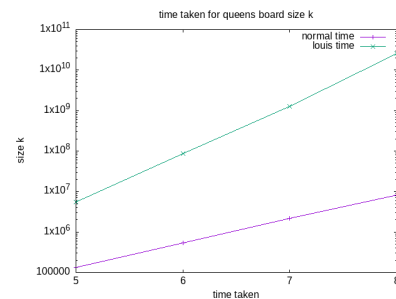
19 (define (test size)
20   (format #t
↪ "~&normal queens x~a: ~a"
↪ size
21         (mattbench (λ() (queens
↪ size)) 10000))
22   (format #t
↪ "~&swapped queens x~a: ~a"
↪ size
23         (mattbench
↪ (λ() (queens-louis size))
↪ 1000)))
24
25 (map (λ(n)
26       (test n))
27       (enumerate-interval 5 8))

```

```

normal queens x5: 135424.6151
swapped queens x5: 5487381.643
normal queens x6: 538798.604
swapped queens x6: 85704466.218
normal queens x7: 2210394.5659
swapped queens x7: 1255288880.717
normal queens x8: 8067290.5992
swapped queens x8: 25384464494.259

```



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

### 3.49 2.2.4: Example: A Picture Language

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

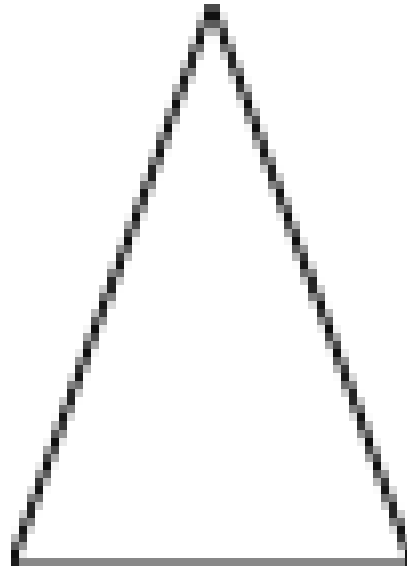


- a **painter** which makes an image within a specified parallelogram shaped frame. This is the most primitive element.

- **Operations** which make new painters from other painters. For example:

- \* *beside* takes two painters, producing a new painter that puts one in the left half and one in the right half.

- \* *flip-horiz* takes one painter and produces another to draw its image right-to-left reversed. These are defined as Scheme procedures and therefore have all the properties of Scheme procedures.



So that works.

How about this?

```
1 (use-modules (pict))
2 (let* ((p (pict-from-file
   ↳ "2/pict/Potato.png"))
3        (pr (rotate p 180)))
4 (pict->file
5  (scale (ht-append (vl-append pr
   ↳ p) pr p (vc-append pr p))
6         0.5)
7  "2/pict/Rotato.svg")) ;; must
   ↳ be SVG
```



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. I installed Guix on my system and ran guix shell guile-picture-language guile. Now the `pict` module is available to me.

```
1 (use-modules (pict))
2 (pict->file (triangle 50 70)
3            "2/pict/test.svg")
```

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

```

1 ;; tilting at windmills trying
  ↳ to make a text-only picture
  ↳ language
2 ;; TODO, probably won't finish
3 (define pl-defaultsize 32)
4 (define (make-painter
  ↳ orientation height width)
5   (cons orientation (cons a b)))
6 (define (p-orientation p)
7   (car p))
8 (define (p-height p)
9   (cadr p))
10 (define (p-width p)
11   (caddr p))
12 (define (opposite orientation)
13   (cond ((eq? orientation 'down)
  ↳ 'up)
14         ((eq? orientation 'up )
  ↳ 'down)
15         ((eq? orientation 'left)
  ↳ 'right)
16         ((eq? orientation
  ↳ 'right) 'left)))
17 (define (flip-p p)
18   (make-painter (opposite
  ↳ (p-orientation p))))
19 (define (below a b)
20   (make-painter 'down a b))
21 (define (beside a b)
22   (make-painter 'right a b))
23 (define (flip-vert p)
24   (if (pair? p)
25       (make-painter ))
26 (define (paint pict)
27   (define (rect p x y)
28     (cond ((string? p)
29           )))
30   (rec p pl-defaultsize
  ↳ pl-defaultsize))
31 (define (beside a b)
32   (hc-append a b))
33 (define (below a b)
34   (vc-append a b))

```

## 3.50 Exercise 2.44: up-split

### 3.50.1 Text Definitions

```

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

```

### 3.50.2 Question

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

### 3.50.3 Answer

```

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

```

## 3.51 Exercise 2.45: Generalized splitting

### 3.51.1 Question

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

```

1 (define right-split (split
  ↳ beside below))

```

```

2 (define up-split (split below
  ↪ beside))

```

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

### 3.51.2 Answer

```

1 (define (split f1 f2)
2   (define (rec painter n)
3     (if (= n 0)
4         painter
5         (let ((smaller (rec
6   ↪ painter
7   ↪ 1))))
8       (f1 painter
9   ↪ (f2 smaller
10  ↪ smaller))))
11 (define right-split (split
  ↪ beside below))
12 (define up-split (split below
  ↪ beside))

```

## 3.52 Exercise 2.46: Defining vectors

### 3.52.1 Question

A two-dimensional vector  $\mathbf{v}$  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

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??*

```

1 ;; Guarantee x is a float
2 (define (float x)
3   (if (inexact? x)
4       x
5       (exact->inexact x)))

```

```

1 <<force-float>>
2 (define (make-vect x y)
3   (cons (float x)
4         (float y))) ;; make-point
5 (define (xcor-vect v)
6   (car v)) ;; x-point
7 (define (ycor-vect v)
8   (cdr v)) ;; y-point
9 (define (add-vect v w)
10  (make-vect (+ (xcor-vect v)
11  ↪ (xcor-vect w))
12  ↪ (+ (ycor-vect v)
13  ↪ (ycor-vect w))))
14 (define (sub-vect v w)
15  (make-vect (- (xcor-vect v)
16  ↪ (xcor-vect w))
17  ↪ (- (ycor-vect v)
18  ↪ (ycor-vect w))))
19 (define (scale-vect s v)
20  (make-vect (* s (xcor-vect v))
21  ↪ (* s (ycor-vect
  ↪ v))))

```

I originally got the definition for `scale-vect` wrong by making it `(scale v s)`, then modified `frame-coord-map`

when that broke. I realized my mistake once I went to write `beside`.

### 3.53 Exercise 2.47: Defining frames

#### 3.53.1 Question

Here are two possible constructors for frames:

```
1 (define (make-frame origin edge1
2   ↪ edge2)
3   (list origin edge1 edge2))
4 (define (make-frame origin edge1
5   ↪ edge2)
6   (cons origin (cons edge1
7   ↪ edge2)))
```

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

#### 3.53.2 Answer

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

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

### 3.54 Exercise 2.48: Line segments

#### 3.54.1 Question

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

#### 3.54.2 Answer

Again reminding me of Exercise 2.2.

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

### 3.55 Exercise 2.49: Primitive painters

#### 3.55.1 Text Definitions

```
1 (define (frame-coord-map frame)
2   (lambda (v)
3     (add-vect
4       (origin-frame frame)
5       (add-vect
6         (scale-vect (xcor-vect v)
7         (edge1-frame
8   ↪ frame))
9       (scale-vect (ycor-vect v)
10      (edge2-frame
11  ↪ frame))))))
12 (define (segments->painter
13   ↪ segment-list)
14   (lambda (frame)
```

```

12 (for-each
13   (lambda (segment)
14     (draw-line
15       ((frame-coord-map frame)
16        (start-segment segment))
17       ((frame-coord-map frame)
18        (end-segment segment))))
19   segment-list)))

```

### 3.55.2 Question

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

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

### 3.55.3 Answer

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

```

1 <<continuous-lines>>
2 (define outline
3   (segments->painter (list
4     (make-segment (make-vect 0 0)
5                   (make-vect 0 1))
6     (make-segment (make-vect 0 1)
7                   (make-vect 1 1))
8     (make-segment (make-vect 1 1)
9                   (make-vect 1 0))
10    (make-segment (make-vect 1 0)
11                  (make-vect 0 0)
12    ↪ 0))))))
12 (define frame-X
13   (segments->painter (list

```

```

14     (make-segment (make-vect 0 0)
15                   (make-vect 1 1))
16     (make-segment (make-vect 0 1)
17                   (make-vect 1 0)
18     ↪ 0))))))
18 (define diamond
19   (segments->painter (list
20     (make-segment (make-vect 0.5
21     ↪ 0)
22                   (make-vect 1
23     ↪ 0.5))
24     (make-segment (make-vect 1
25     ↪ 0.5)
26                   (make-vect 0.5
27     ↪ 1))
28     (make-segment (make-vect 0.5
29     ↪ 1)
30                   (make-vect 0
31     ↪ 0.5))
32     (make-segment (make-vect 0
33     ↪ 0.5)
34                   (make-vect 0.5
35     ↪ 0))))))

```

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

```

1 (use-modules (pict))
2 <<make-frame>>
3 <<make-vect>>
4 <<make-segment>>
5 (define (frame-coord-map frame)
6   ;; Returns a function for
7   ↪ adjusting a frame by a vector
8   (lambda (v)
9     (add-vect
10      (origin-frame frame)
11      (add-vect
12       (scale-vect (xcor-vect v)
13                  (edge1-frame
14     ↪ frame)))
15      (scale-vect (ycor-vect v)
16                  (edge2-frame
17     ↪ frame))))))
15 (define (draw-line start end)

```

```

16  ;; take two vectors, returns a
    ↳ line SVG object for pict
17  (line (xcor-vect start)
18        (ycor-vect start)
19        (xcor-vect end)
20        (ycor-vect end)))
21  (define (segments->painter
    ↳ segment-list)
22  ;; takes a list of segments,
    ↳ returns a "painter" lambda,
    ↳ which applies a frame
23  ;; to those segments and then
    ↳ maps over the result with
    ↳ draw-line to make a
24  ;; list of SVG line objects
    ↳ which pict can combine.
25  (lambda (frame)
26    (map
27      (lambda (segment)
28        (draw-line
29          ((frame-coord-map frame)
30           (start-segment segment))
31          ((frame-coord-map frame)
32           (end-segment segment))))
33      segment-list)))
34  ;; NOTE: in the text, draw-line
    ↳ is a function which triggers
    ↳ an action in
35  ;; some graphics driver, and
    ↳ returns nothing. Because of
    ↳ this, (map) was
36  ;; originally (for-each). Thus
    ↳ the final result would have
    ↳ been thrown away.

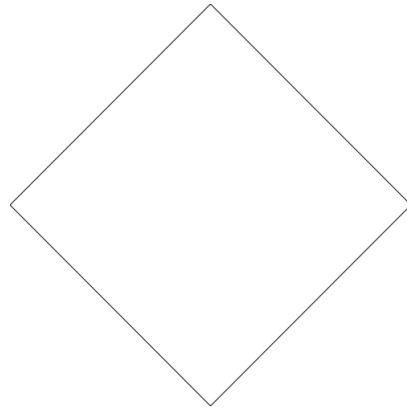
37  <<painters>>
38
39  (define (paint-lines painter)
40  ;; use pict to compile an SVG
    ↳ with the elements described
    ↳ by painter
41  (let ((Frame (make-frame
42              (make-vect 0 0)
43              (make-vect 500 0)
44              (make-vect 0 500))))
45    (apply lt-superimpose
46      (painter Frame))))

```

```

1  <<linepainter-pict>>
2  (pict->file (paint-lines diamond)
3
    ↳ "2/pict/testline.svg")
4  ; #:maxw 100 #:maxh 100) <-
    ↳ FIXME: This procedure should
    ↳ take these arguments
5  ; but I can't get it to work.

```



Holy moly it actually works.

## 3.56 Exercise 2.50: Transforming painters

### 3.56.1 Text Definitions

```

1  (define (transform-painter
2    painter origin corner1
3    ↳ corner2)
4    (lambda (frame)
5      (let ((m (frame-coord-map
6                ↳ frame)))
7        (let ((new-origin (m
8                            ↳ origin)))
9          (painter (make-frame
10                    ↳ new-origin
11                    ↳ (sub-vect (m corner1)
12                                ↳ new-origin)
13                    ↳ (sub-vect (m corner2)

```

```

10 ↪      new-origin)))))))))
11
12 (define (flip-vert painter)
13   (transform-painter
14     painter
15     (make-vect 0.0 1.0) ; new
16     ↪ origin
17     (make-vect 1.0 1.0) ; new
18     ↪ end of edge1
19     (make-vect 0.0 0.0))) ; new
20     ↪ end of edge2
21
22 (define (rotate90 painter)
23   (transform-painter painter
24     (make-vect
25       ↪ 1.0 0.0)
26       (make-vect
27       ↪ 1.0 1.0)
28       (make-vect
29       ↪ 0.0 0.0)))
30
31 (define (squash-inwards painter)
32   (transform-painter painter
33     (make-vect
34       ↪ 0.0 0.0)
35       (make-vect
36       ↪ 0.65 0.35)
37       (make-vect
38       ↪ 0.35 0.65)))
39
40 (define (beside painter1
41   ↪ painter2)
42   (let ((split-point (make-vect
43     ↪ 0.5 0.0)))
44     (let ((paint-left
45       ↪ (transform-painter
46         painter1
47         ↪ (make-vect 0.0 0.0)
48         ↪ split-point
49         ↪ (make-vect 0.0 1.0)))
50         (paint-right
51       ↪ (transform-painter
52         painter2
53         ↪ split-point
54         ↪ (make-vect 0.0 1.0)
55         ↪ (transform-painter
56           painter2
57           ↪ split-point

```

```

40 ↪ (make-vect 1.0 0.0)
41
42 ↪ (make-vect 0.5 1.0)))
43   (lambda (frame)
44     (append
45       (paint-left frame)
46       (paint-right frame))))))

```

```

1 <<linepainter-pict>>
2 <<frame-transforms-txt>>
3 (define topleft-tri
4   (segments->painter (list
5     ↪ (make-segment (make-vect 0 0)
6     ↪ (make-vect 0 0.4))
7     ↪ (make-segment (make-vect 0
8     ↪ 0.4)
9     ↪ (make-vect 0.4 0))
10    ↪ (make-segment (make-vect 0.4
11    ↪ 0)
12    ↪ (make-vect 0 0))))))
13 (let ((picture (beside (beside
14   ↪ topleft-tri
15   ↪ topleft-tri)
16     (rotate90
17     ↪ topleft-tri))))
18   (pict->file (paint-lines picture)
19     ↪ "2/pict/rotatetest.svg"))

```



### 3.56.2 Question

Define the transformation `flipip-horiz`, which flips painters

horizontally, and transformations that rotate painters counterclockwise by 180 degrees and 270 degrees.

### 3.56.3 Answer

```

1 (define (flip-horiz painter)
2   (transform-painter
3     painter
4     (make-vect 1.0 0.0) ; new
    ↪ origin
5     (make-vect 0.0 0.0) ; new
    ↪ end of edge1
6     (make-vect 1.0 1.0))) ; new
    ↪ end of edge2
7
8 (define (rotate180 painter)
9   (transform-painter painter
10                      (make-vect
11 ↪ 1.0 1.0)
12                      (make-vect
13 ↪ 0.0 1.0)
14                      (make-vect
15 ↪ 1.0 0.0)))
16 (define (rotate270 painter)
17   (transform-painter painter
18                      (make-vect
19 ↪ 0.0 1.0)
20                      (make-vect
21 ↪ 0.0 0.0)
22                      (make-vect
23 ↪ 1.0 1.0)))

```

```

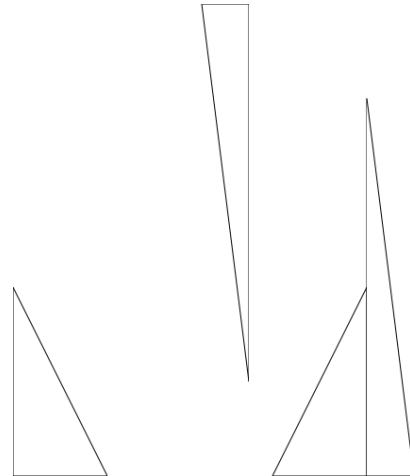
1 <<linepainter-pict>>
2 <<frame-transforms-txt>>
3 <<flip-rotate2>>
4 (define topleft-tri
5   (segments->painter (list
6 ↪ (make-segment (make-vect 0 0)
7 ↪ (make-vect 0 0.4))
8 ↪ (make-segment (make-vect 0
9 ↪ 0.4)

```

```

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

```



## 3.57 Exercise 2.51

### 3.57.1 Question

Define the `below` operation for painters. `below` takes two painters as arguments. The resulting painter, given a frame, draws with the first painter in the bottom of the



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

### 3.57.2 Answer

```

1 (define (below painter1 painter2)
2   (let ((split-point (make-vect
3     ↪ 0.0 0.5)))
4     (let ((paint-left
5       ↪ (transform-painter
6         ↪ painter1
7         ↪ (make-vect 0.0 0.0)
8         ↪ (make-vect 1.0 0.0)
9         ↪ split-point))
10      (paint-right
11        ↪ (transform-painter
12          ↪ painter2
13          ↪ split-point
14          ↪ (make-vect 1.0 0.5)
15          ↪ (make-vect 0.0 1.0))))))
16   (lambda (frame)
17     (append
18       (paint-left frame)
19       (paint-right frame))))))
17 (define (below-rotate painter1
18   ↪ painter2)
19   (rotate270 (beside
20     ↪ (rotate90 painter2)
21     ↪ painter1))))

```

```

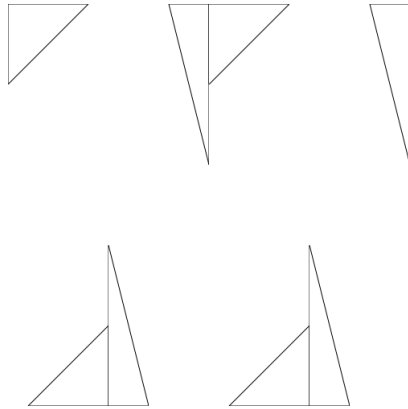
1 <<linepainter-pict>>
2 <<frame-transforms-txt>>
3 <<flip-rotate2>>

```

```

4 <<below>>
5 (define topleft-tri
6   (segments->painter (list
7     ↪ (make-segment (make-vect 0 0)
8     ↪ (make-vect 0 0.4))
9     ↪ (make-segment (make-vect 0
10      ↪ 0.4)
11      ↪ (make-vect 0.8 0))
12     ↪ (make-segment (make-vect 0.8
13      ↪ 0)
14      ↪ (make-vect 0 0))))))
15 (let ((p1 (below (beside
16   ↪ topleft-tri
17   ↪ (rotate90 topleft-tri)
18   ↪ (beside
19     ↪ (rotate180 topleft-tri)
20     ↪ (rotate270 topleft-tri))))))
21   (p2 (below-rotate (beside
22     ↪ topleft-tri
23     ↪ (rotate90 topleft-tri)
24     ↪ (beside
25       ↪ (rotate180 topleft-tri)
26       ↪ (rotate270 topleft-tri))))))
27   (pict->file (paint-lines
28     ↪ (beside p1 p2))
29     ↪ "2/pict/rotate3.svg"))

```



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

### 3.58 2.2.4 continued

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

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

### 3.59 Exercise 2.52

#### 3.59.1 Question A

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

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

#### 3.59.2 Answer

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.

```

1 (define (continuous-lines .
2   ↪ vectors)
3   ;; Given a list of vectors,
4   ↪ return a series of line
5   ↪ segments
6   ;; that continuously follow
7   ↪ the vectors until there are
8   ↪ no more,
9   ;; at which point connect the
10  ↪ last vector to the first.
11  (define (iter vecs lines)
12    (if (> 2 (length vecs))
13        lines
14        (let* ((first (car vecs))
15               (rest (cdr vecs))
16               (second (car
17 ↪ rest)))
18          (new-line
19 ↪ (make-segment first second)))
20    (iter rest (cons
21 ↪ new-line lines))))
22  (let ((input (if (= 1 (length
23 ↪ vectors))
24                  (car vectors)
25 ↪ ; assume we were passed a
26 ↪ list, unwrap it
27                  vectors)))
28    (iter (append input
29 ↪ (list (car
30 ↪ input))) ; Connect to start
31          '()))

```

```

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

```

