

A Journey Through SICP

Notes, exercises and analyses of Abelson and Sussman

ProducerMatt

September 30, 2022

Contents

1	Introduction Notes	5
1.1	Text Foreword	5
1.2	Preface, 1e	6
2	Chapter 1: Building Abstractions with Procedures	7
2.1	1.1: The Elements of Programming	7
2.2	1.1.1: Expressions	8
2.3	1.1.3: Evaluating Combinations	8
2.4	1.1.4: Compound Procedures	9
2.5	1.1.5: The Substitution Model for Procedure Application	9
2.6	1.1.6: Conditional Expressions and Predicates	10
2.7	Exercise 1.1	11
	2.7.1 Question	11
	2.7.2 Answer	11
2.8	Exercise 1.2	12
	2.8.1 Question	12
	2.8.2 Answer	12
2.9	Exercise 1.3	12
	2.9.1 Question	12
	2.9.2 Answer	12
2.10	Exercise 1.4	13
	2.10.1 Question	13
	2.10.2 Answer	13
2.11	Exercise 1.5	13
	2.11.1 Question	13
	2.11.2 Answer	14
2.12	1.1.7: Example: Square Roots by Newton's Method	14
2.13	1.1.8: Procedures as Black-Box Abstractions	14
2.14	Exercise 1.6	15
	2.14.1 Text code	15
	2.14.2 Question	15
	2.14.3 Answer	16
2.15	Exercise 1.7	16
	2.15.1 Text	16
	2.15.2 Question	16
	2.15.3 Diary	17
	2.15.4 Answer	19
2.16	Exercise 1.8	20
	2.16.1 Question	20
	2.16.2 Diary	20
	2.16.3 Answer	21
2.17	1.2: Procedures and the Processes They Generate	22
2.18	1.2.1: Linear Recursion and Iteration	22
2.19	Exercise 1.9	23

2.19.1	Question	23
2.19.2	Answer	23
2.20	Exercise 1.10	24
2.20.1	Question	24
2.20.2	Answer	24
2.21	1.2.2: Tree Recursion	26
2.21.1	Example: Counting change	26
2.22	Exercise 1.11	26
2.22.1	Question	26
2.22.2	Answer	27
2.23	Exercise 1.12	29
2.23.1	Question	29
2.23.2	Answer	29
2.24	Exercise 1.13	30
2.24.1	Question	30
2.24.2	Answer	30
2.25	1.2.3: Orders of Growth	32
2.26	Exercise 1.14	32
2.26.1	Question A	33
2.26.2	Answer	33
2.26.3	Question B	36
2.26.4	Answer B	36
2.27	Exercise 1.15	39
2.27.1	Question A	39
2.27.2	Answer A	40
2.27.3	Question B	40
2.27.4	Answer B	40
2.28	Exercise 1.16	43
2.28.1	Text	43
2.28.2	Question	44
2.28.3	Diary	44
2.28.4	Answer	45
2.29	Exercise 1.17	46
2.29.1	Question	46
2.29.2	Answer	47
2.30	Exercise 1.18	47
2.30.1	Question	47
2.30.2	Diary	47
2.30.3	Answer	48
2.31	Exercise 1.19	49
2.31.1	Question	49
2.31.2	Diary	50
2.31.3	Answer	51
2.32	1.2.5: Greatest Common Divisor	52
2.33	Exercise 1.20	52
2.33.1	Text	52

2.33.2	Question	53
2.33.3	Answer	53
2.34	1.2.6: Example: Testing for Primality	56
2.35	Exercise 1.21	56
2.35.1	Text	56
2.35.2	Question	57
2.35.3	Answer	57
2.36	Exercise 1.22	57
2.36.1	Question	57
2.36.2	Answer	58
2.37	Exercise 1.23	61
2.37.1	Question	61
2.37.2	A Comedy of Error (just the one)	62
2.37.3	Answer	64
2.38	Exercise 1.24	68
2.38.1	Text	68
2.38.2	Question	68
2.38.3	Answer	68
2.39	Exercise 1.25	72
2.39.1	Question	72
2.39.2	Answer	72
2.40	Exercise 1.26	73
2.40.1	Question	73
2.40.2	Answer	74
2.41	Exercise 1.27	74
2.41.1	Question	74
2.41.2	Answer	74
2.42	Exercise 1.28	75
2.42.1	Question	75
2.42.2	Analysis	75
2.42.3	Answer	77
2.43	1.3: Formulating Abstractions with Higher-Order Procedures	79
2.44	1.3.1: Procedures as Arguments	79
2.45	Exercise 1.29	80
2.45.1	Text	80
2.45.2	Question	80
2.45.3	Answer	81
2.46	Exercise 1.30	82
2.46.1	Question	82
2.46.2	Answer	82
2.47	Exercise 1.31	82
2.47.1	Question A.1	82
2.47.2	Answer A.1	82
2.47.3	Question A.2	82
2.47.4	Answer A.2	83
2.47.5	Question A.3	83

2.47.6	Answer A.3	83
2.47.7	Question B	83
2.47.8	Answer B	84
2.48	Exercise 1.32	84
2.48.1	Question A	84
2.48.2	Answer A	85
2.48.3	Question B	86
2.48.4	Answer B	86
2.49	Exercise 1.33	86
2.49.1	Question A	86
2.49.2	Answer A	86
2.49.3	Question B	86
2.50	1.3.2: Constructing Procedures Using lambda	88
2.51	Exercise 1.34	88
2.51.1	Question	88
2.51.2	Answer	88
2.52	1.3.3 Procedures as General Methods	89
2.53	Exercise 1.35	89
2.53.1	Text	89
2.53.2	Question	90
2.53.3	Answer	90
2.54	Exercise 1.36	90
2.54.1	Question	90
2.54.2	Answer	90
2.55	Exercise 1.37	91
2.55.1	Question A	91
2.55.2	Answer A	92
2.55.3	Question B	92
2.55.4	Answer B	92
2.55.5	Question C	93
2.55.6	Answer C	93
2.56	Exercise 1.38	94
2.56.1	Question	94
2.56.2	Answer	94
2.57	Exercise 1.39	94
2.57.1	Question	94
2.57.2	Answer	95
2.58	1.3.4 Procedures as Returned Values	95
2.59	Exercise 1.40	95
2.59.1	Text	95
2.59.2	Question	96
2.59.3	Answer	96
2.60	Exercise 1.41	97
2.60.1	Question	97
2.60.2	Answer	97
2.61	Exercise 1.42	97

2.61.1	Question	97
2.61.2	Answer	97
2.62	Exercise 1.43	98
2.62.1	Question	98
2.62.2	Answer	98
2.63	Exercise 1.44	98
2.63.1	Question	98
2.63.2	Answer	99
2.64	Exercise 1.45	99
2.64.1	Question	99
2.64.2	Answer	99
2.65	Exercise 1.46	102
2.65.1	Question	102
2.65.2	Answer	102
3	Chapter 2: Building Abstractions with Data	104
3.1	2.1.1: Example: Arithmetic Operations for Rational Numbers	104
3.2	Exercise 2.1	104
3.2.1	Text	104
3.2.2	Question	105
3.2.3	Answer	105
3.3	Exercise 2.2	107
3.3.1	Question	107
3.3.2	Answer	107
3.4	Exercise 2.3	108
3.4.1	Question	108
3.4.2	Answer 1	109
3.4.3	Answer 2	112
3.5	2.1.3: What Is Meant by Data?	115
3.6	2.2: Hierarchical Data and the Closure Property	115
3.7	2.2.3: Sequences as Conventional Interfaces	116
3.8	2.2.4: Example: A Picture Language	118

1 Introduction Notes

1.1 Text Foreword

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

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

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

1.2 Preface, 1e

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

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

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

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

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

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

1. Techniques for controlling complexity

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

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

One way to find \sqrt{x} is to take our function for approaching a square root ($\lambda(\text{guess target}) (\text{average guess (divide target guess)})$), 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.

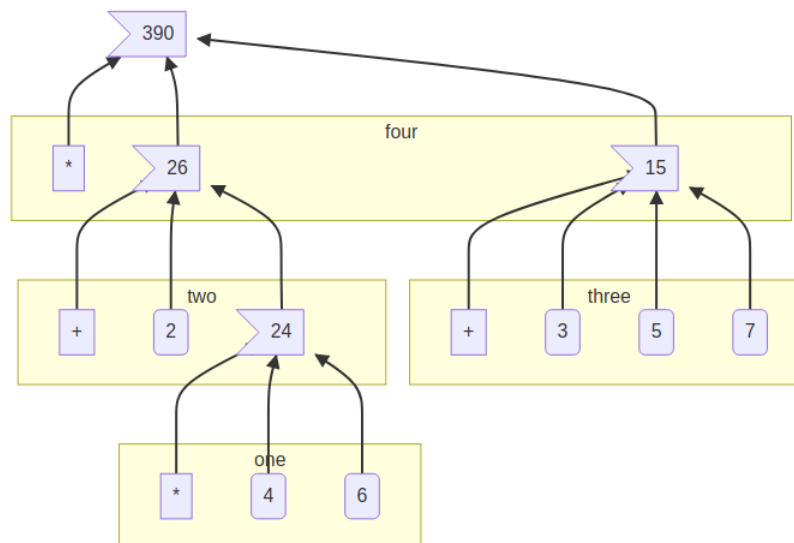
Lisp uses **prefix notation**, which is not customary mathematical notation, but provides several advantages.

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.

2.3 1.1.3: Evaluating Combinations

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

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



2.4 1.1.4: Compound Procedures

We have identified the following in Lisp:

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

Procedure definitions give a name to a compound procedure.

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

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

2.5 1.1.5: The Substitution Model for Procedure Application

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

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

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

Evaluations can be done in different orders.

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

Normal order substitutes expressions until it obtains an expression involving

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

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

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

```

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

```

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

2.6 1.1.6: Conditional Expressions and Predicates

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

```

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

```

A shorter form of conditional:

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

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

```
1 (and expression1 ... expressionN)
2 ; if encounters false, stop eval and returns false.
3 (or expression1 ... expressionN)
4 ; if encounters true, stop eval and return true. Else false.
5 (not expression)
6 ; true if 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 b))
6                  ((and (>= a b) (> c b)) (sum-square a c))
7                  (else (sum-square b c))))

```

```

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

```

```

(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?

2.15.3 Diary

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

```

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

```

This solution can correctly find small and large numbers:

```

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

```

3162277.6601683795

```

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

```

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

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

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

2. 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)) ;; improve call 1
9 (define (sqrt-iter guess x)
10   (if (good-enough? guess x)
11       guess
12       (sqrt-iter (improve guess x) x))) ;; call 2
13 (define (sqrt x)
14   (sqrt-iter 1.0 x))
15 (newline)
16 (display (mattbench (lambda () (sqrt 69420)) 400000000))
17 (newline)
18 ;; 4731.30 <- Benchmark results
```

(b) Optimized

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

```

15 (sqrt-iter 1.0 x))
16 (newline)
17 (display (mattbench (λ() (sqrt 69420)) 400000000))
18 (newline)

```

(c) Benchmark results

Unoptimized	4731.30
Optimized	2518.44

2.15.4 Answer

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

```

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

```

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

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

```

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

```

1000000.0

So, my definition of sqrt:

```

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

```

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

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

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

2.16 Exercise 1.8

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:

$$\frac{\frac{x}{y^2} + 2y}{3} \quad (1)$$

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

2.16.2 Diary

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

```
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 (cb-iter guess x counter)
12   (if (or (cb-good-enough? guess x) (> counter 100))
13       guess
14       (begin
```

```

15      (cbrt-iter (cb-improve guess x) x (+ 1 counter))))))
16 (define (cbrt x)
17   (cbrt-iter 1.0 x 0))
18
19 (try-these cbrt 7 32 56 100)

```

```

      7  1.912931182772389
     32  3.174802103936399
     56  3.825862365544778
    100  4.641588833612779

```

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

2.16.3 Answer

```

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

```

```

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

```

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

2.17 1.2: Procedures and the Processes They Generate

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

2.18 1.2.1: Linear Recursion and Iteration

Consider these two procedures for obtaining factorials:

```

1 (define (factorial-recursion n)
2   (if (= n 1)
3       1
4       (* n
5         (factorial-recursion (- n 1)))))
6
7 (define (factorial-iteration n)
8   (define (fact-iter product counter max-count)
9     (if (> counter max-count)
10        product
11        (fact-iter
12          (* counter product)
13          (+ counter 1)
14          max-count)))
15
16 (fact-iter 1 1 n))

```

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

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

recursive procedure is a procedure whose definition refers to itself.

recursive process is a process that evolves recursively.

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

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

2.19 Exercise 1.9

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.

Pretend there's a Pascal's triangle here.

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

2.23.2 Answer

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

```
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

2.24.1 Question

Prove that $\text{Fib}(n)$ is the closest integer to $\frac{\phi^n}{\sqrt{5}}$ where ϕ is $\frac{1+\sqrt{5}}{2}$.

Hint: let $\psi = \frac{1-\sqrt{5}}{2}$. Use induction and the definition of the Fibonacci numbers to prove that

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

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

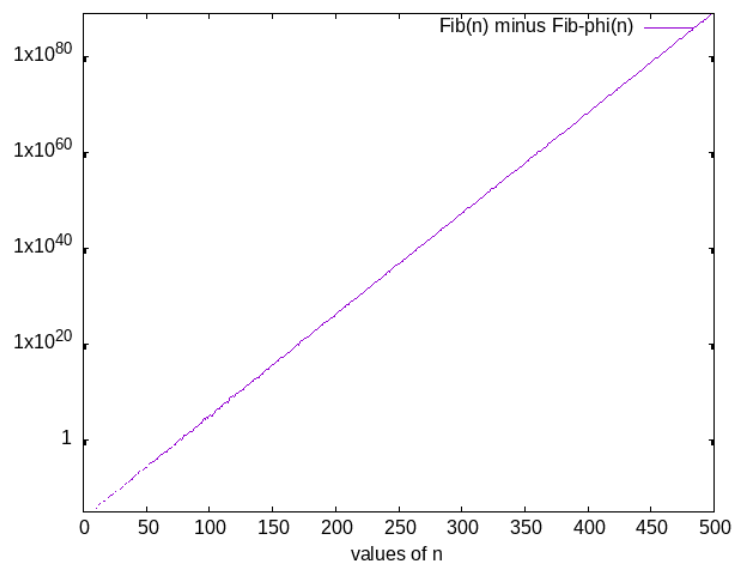
```

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 Υ is the golden ratio $\frac{1+\sqrt{5}}{2}$

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

2.26 Exercise 1.14

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

```
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)
18         ((= kinds-of-coins 3) 10)
19         ((= kinds-of-coins 4) 25))
```

```
20 ((= kinds-of-coins 5) 50)))
```

2.26.1 Question A

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

2.26.2 Answer

I want to generate this graph algorithmically.

```
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: https://stackoverflow.com/a/14806144
13  (define (cc amount kinds-of-coins oldbubble)
14    (let ((recur (lambda (new-amount new-kinds)
15                  (begin
16                    (display "\n") ;; Source bubble
17                    (display `(\n" (oldbubble ,amount ,kinds-of-coins)
18                    (display "\n")
19                    (display " -> ") ;; arrow pointing from
20                    (display "\n") ;; child bubble
21                    (display `(\n" (bubblecounter ,new-amount
22                    (display "\n")
23                    (display "\n")
24                    (cc new-amount new-kinds bubblecounter))))))
25    (set! bubblecounter (+ bubblecounter 1))
26    (cond ((= amount 0) 1)
27          ((or (< amount 0) (= kinds-of-coins 0)) 0)
28          (else (+
29                 (recur amount (- kinds-of-coins 1))
30                 (recur (- amount
31                        (first-denomination kinds-of-coins))
```

```

32         kinds-of-coins))))))
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.

```

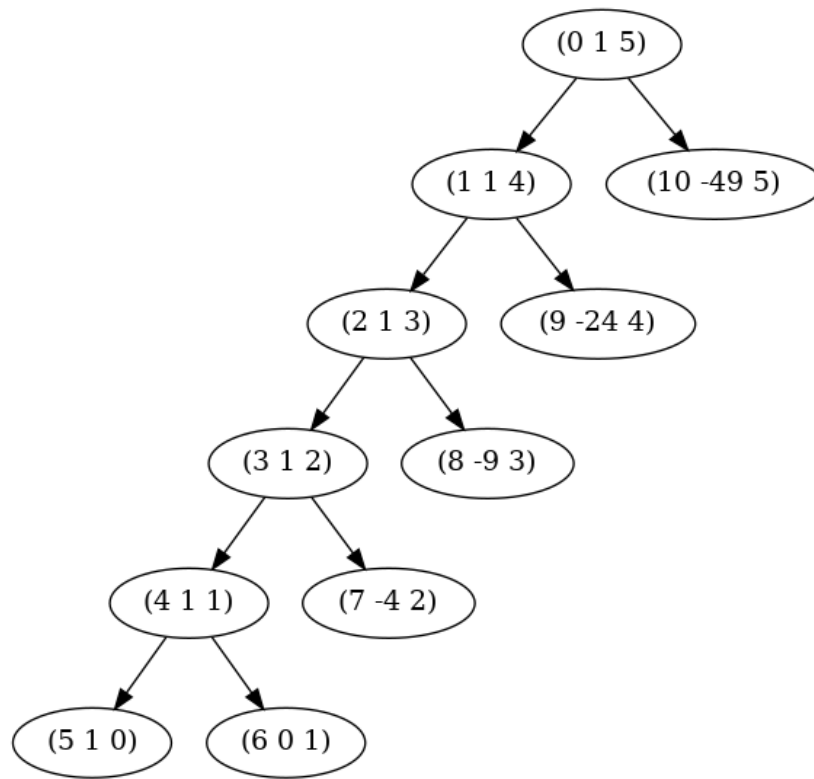
1 <<count-change-graphviz>>
2 (count-change 1)

```

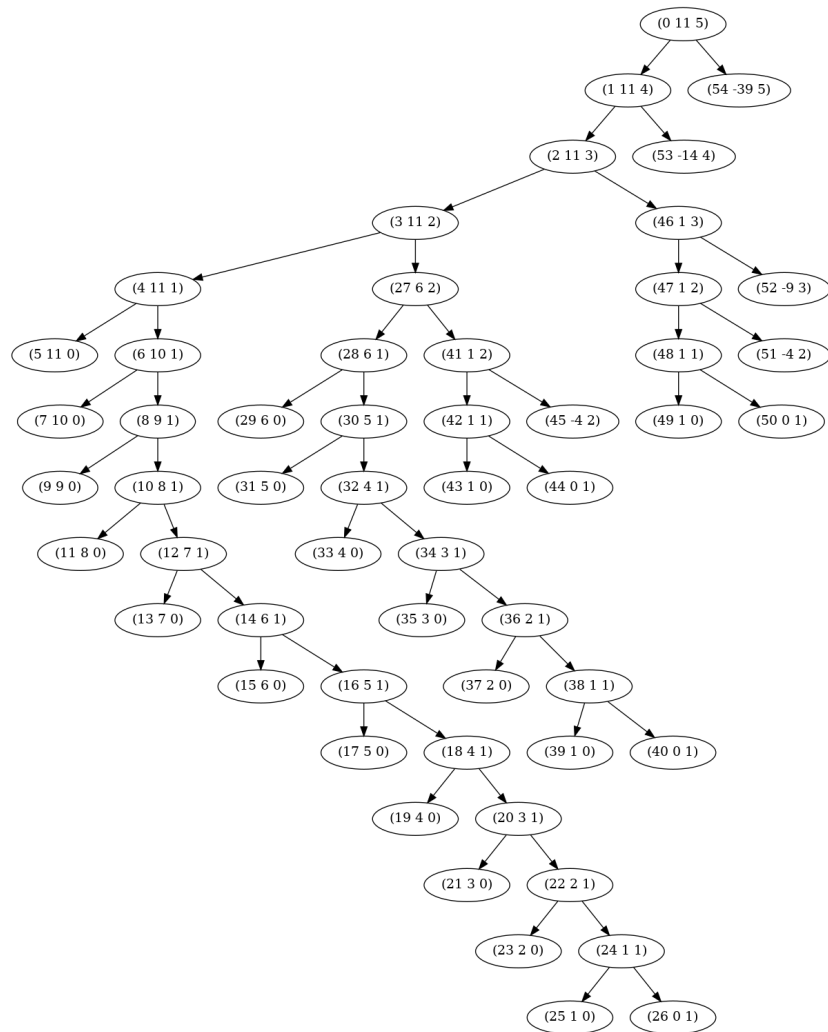
```

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:



2.26.3 Question B

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

2.26.4 Answer B

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

```

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 Θ notation in Section 1.2.3. However, my lack of formal math education made realizing the significance of this passage difficult. For one, I didn't understand that $k_1f(n) \leq R(n) \leq k_2f(n)$ means "you can find the Θ by proving that a graph of the algorithm's resource usage is bounded by two identical functions multiplied by constants." So, the graph of resource usage for an algorithm with $\Theta(n^2)$ will 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 Θ and Ω and O notations (they are all different btw!) is about aggressively simplifying to make a very general statement about the behavior of the algorithm.

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

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

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

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

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

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

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

2.27 Exercise 1.15

2.27.1 Question A

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

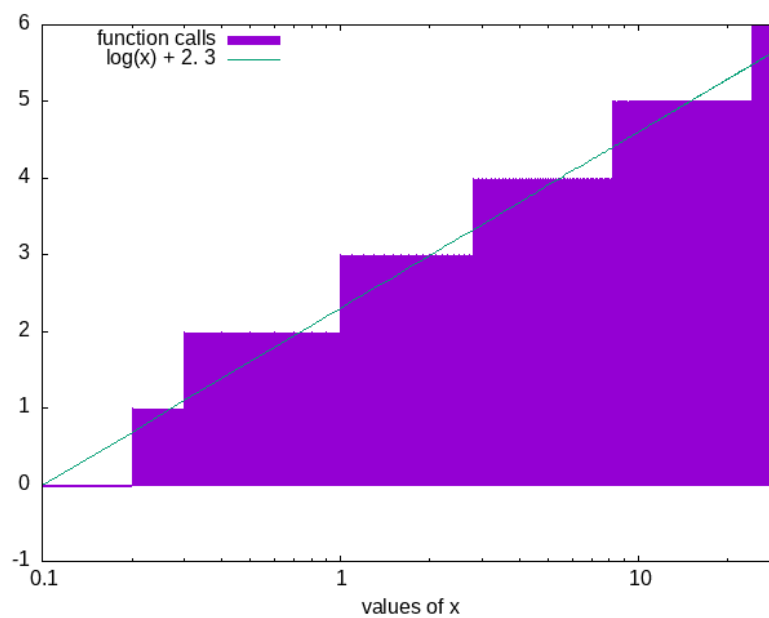
2.27.4 Answer B

```
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 0.05))))))
8 (p (p (p (p 0.1495000000000002))))
9 (p (p (p 0.4351345505000005)))
10 (p (p 0.9758465331678772))
11 (p -0.7895631144708228)
12 -0.39980345741334

```

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

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

```

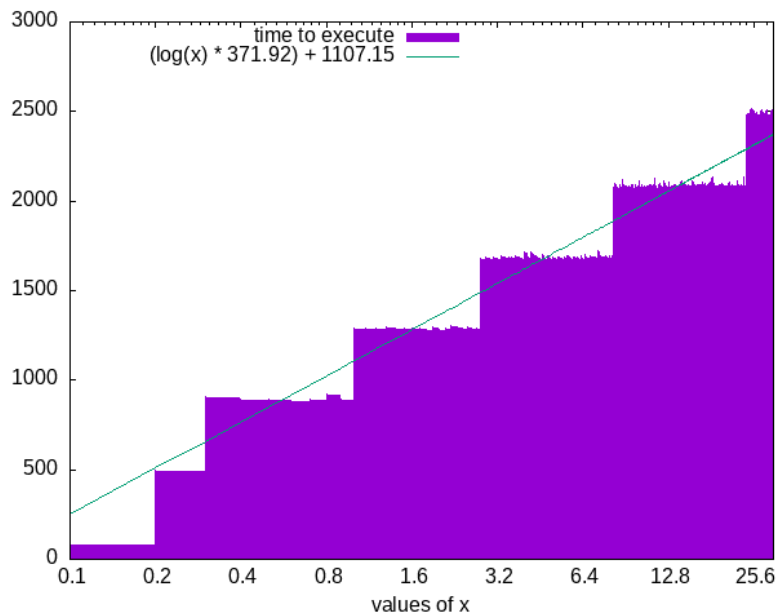
1 ;; This execution takes too long for org-mode, so I'm doing it
2 ;; externally and importing the results
3 (use-srfis '(1))
4 (use-modules (ice-9 format))
5 (load "../mattbench.scm")
6 <<1-15-deps>>
7 (let* ((vals (iota 300 0.1 0.1))
8         (times (map (lambda (i)
9                       (mattbench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

```

```

1 reset # helps with various issues in execution
2 set xtics 0.5
3 set xlabel 'values of x'
4 set logscale x
5 set key top left
6 set style fill solid 1.00 border
7 #set style function fillsteps below
8
9 f(x) = (log(x) * a) + b
10 fit f(x) 'Ex15/sine-bench.dat' using 1:2 via a,b
11
12 plot 'Ex15/sine-bench.dat' using 1:2 with fillsteps title
13   'time to execute', \
14   'Ex15/sine-bench.dat' using 1:(f($1)) with lines title
15   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 (cond ((= n 1) (begin (format #t "~&|~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" "variable")
12 (format #t "~&|--|--|--|~%" )
13 (iter b n 1))

```

```

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

```

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

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

2.28.4 Answer

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

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

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 (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)))) ; This was kind of
9   ↪ a stretch to think of.G
10          ;(fast-mult (double a) (halve b))) <== My first
11   ↪ instinct is iterative
12   (else (+ a (fast-mult-rec a (- b 1))))))
```

Proof it works:

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

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

2.30 Exercise 1.18

2.30.1 Question

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

2.30.2 Diary

1. Comparison benchmarks:


```

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

```

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

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

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

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

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

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

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:

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

Call this transformation T , and observe that applying T over and over again n times, starting with 1 and 0, produces the pair $\text{__Fib}_{(n+1)}$ 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 $a < -bq + aq + ap$ and $b < -bp + aq$. Show that if we apply such a transformation $T_{(pq)}$ twice, the effect is the same as using a single transformation $T_{(p'q')}$ of the same form, and compute p' and q' in terms of p and q . This gives us an explicit way to square these transformations, and thus we can compute T^n using successive squaring, as in the ‘fast-expt’ procedure. Put this all together to complete the following procedure, which runs in a logarithmic number of steps:

```

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))
13                    (+ (* b p) (* a q))
14                    p
15                    q
16                    (- count 1))))))

```

2.31.2 Diary

More succinctly put:

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

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

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

```

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 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 p'
16                    (+ (* p q)
17                      (* q q)) ; compute q'
18                    (/ count 2)))
19         (else (fib-iter (+ (* b q) (* a q) (* a p))
20                          (+ (* b p) (* a q))
21                          p
22                          q
23                          (- count 1)))))
24

```

“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.

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

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

2.33 Exercise 1.20

2.33.1 Text

```

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 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 40))))
17
18  ; evaluate 1 remainder
19  (if (= 6 0)
20      40
21      (gcd (remainder 206 40)
22            (remainder 40 (remainder 206 40))))
23
24  ; recurse
25  (gcd (remainder 206 40)
26        (remainder 40 (remainder 206 40)))
27
28  ; encounter conditional
29  (if (= (remainder 40 (remainder 206 40)) 0)
30      (remainder 206 40)
31      (gcd (remainder 40 (remainder 206 40))
32            (remainder (remainder 206 40) (remainder 40
33  ↪ (remainder 206 40)))))
34
35  ; eval 2 remainder
36  (if (= 4 0)
37      (remainder 206 40)
38      (gcd (remainder 40 (remainder 206 40))
39            (remainder (remainder 206 40) (remainder 40
40  ↪ (remainder 206 40)))))
41
42  ; recurse
43  (gcd (remainder 40 (remainder 206 40))
44        (remainder (remainder 206 40) (remainder 40 (remainder
45  ↪ 206 40)))))
46
47  ; encounter conditional
48  (if (= (remainder (remainder 206 40) (remainder 40
49  ↪ (remainder 206 40))) 0)
50      (remainder 40 (remainder 206 40))
51      (gcd (remainder (remainder 206 40) (remainder 40
52  ↪ (remainder 206 40)))

```

```

48      (remainder (remainder 40 (remainder 206 40))
↳ (remainder (remainder 206 40) (remainder 40 (remainder
↳ 206 40))))))
49
50 ; eval 4 remainders
51 (if (= 2 0)
52     (remainder 40 (remainder 206 40))
53     (gcd (remainder (remainder 206 40) (remainder 40
↳ (remainder 206 40)))
54         (remainder (remainder 40 (remainder 206 40))
↳ (remainder (remainder 206 40) (remainder 40 (remainder
↳ 206 40))))))
55
56 ; recurse
57 (gcd (remainder (remainder 206 40) (remainder 40 (remainder
↳ 206 40)))
58     (remainder (remainder 40 (remainder 206 40))
↳ (remainder (remainder 206 40) (remainder 40 (remainder
↳ 206 40))))))
59
60 ; encounter conditional
61 (if (= (remainder (remainder 40 (remainder 206 40))
↳ (remainder (remainder 206 40) (remainder 40 (remainder
↳ 206 40)))) 0)
62     (remainder (remainder 206 40) (remainder 40 (remainder
↳ 206 40)))
63     (gcd (remainder (remainder 40 (remainder 206 40))
↳ (remainder (remainder 206 40) (remainder 40 (remainder
↳ 206 40)))) (remainder a (remainder (remainder 40
↳ (remainder 206 40)) (remainder (remainder 206 40)
↳ (remainder 40 (remainder 206 40))))))
64
65 ; eval 7 remainders
66 (if (= 0 0)
67     (remainder (remainder 206 40) (remainder 40 (remainder
↳ 206 40)))
68     (gcd (remainder (remainder 40 (remainder 206 40))
↳ (remainder (remainder 206 40) (remainder 40 (remainder
↳ 206 40)))) (remainder a (remainder (remainder 40
↳ (remainder 206 40)) (remainder (remainder 206 40)
↳ (remainder 40 (remainder 206 40))))))
69
70 ; eval 4 remainders

```



```

71 (remainder (remainder 206 40) (remainder 40 (remainder 206
    ↪ 40)))
72 ; => 2

```

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

2.34 1.2.6: Example: Testing for Primality

Two algorithms for testing primality of numbers.

1. $\Theta(\sqrt{n})$: Start with $x = 2$, check for divisibility with n , if not then increment x by 1 and check again. If $x^2 > n$ and you haven't found a divisor, n is prime.
2. $\Theta(\log n)$: Given a number n , pick a random number $a < n$ and compute the remainder of a^n modulo n . If the result is not equal to a , then n is certainly not prime. If it is a , then chances are good that n is prime. Now pick another random number a and test it with the same method. If it also satisfies the equation, then we can be even more confident that n is prime. By trying more and more values of a , we can increase our confidence in the result. This algorithm is known as the Fermat test.

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

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

2.35 Exercise 1.21

2.35.1 Text

```

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 (- (get-internal-run-time)
10                          start-time))
11         n)

```

```

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

```

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

2.36.2 Answer

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

```

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

```

```

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

```

```

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

```

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

```

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

```

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

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

```

1 (define (mattbench2 f n)
2   ;; Executes "f" for n times, and returns how long it took.
3   ;; f is a lambda that takes no arguments, a.k.a. a "thunk"
4
5   ;; Returns a list with car(last execution results) and
6   ↪ cadr(time taken divided by iterations n)
7
8   (define (time-getter) (get-internal-run-time))
9   (define start-time (time-getter))
10  (define (how-long) (- (time-getter) start-time))

```

```

11 (define (iter i)
12   (f)
13   (if (<= i 0)
14       (f) ;; return the results of the last function call
15       (iter (- i 1))))
16
17 (list (iter n) ;; result of last call of f
18       (/ (how-long) (* n 1.0))));; Divide by iterations
↪ so changed n has no effect

```

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

```

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

```

I can benchmark these functions like so:

```

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

```

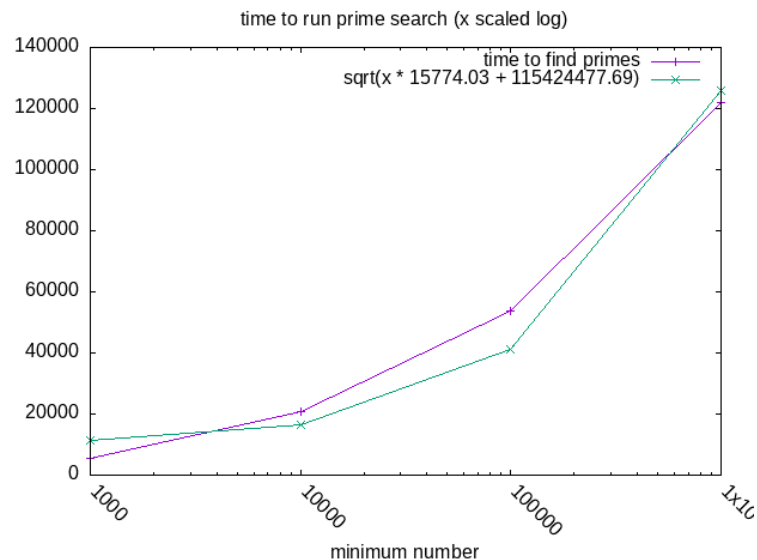
```

12 (print-row
13 (testit search-for-primes))
14

```

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

5425.223086 20772.332491 53577.240193 121986.712395



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

2.37 Exercise 1.23

2.37.1 Question

The `smallest-divisor` procedure shown at the start of this section does lots of needless testing: After it checks to see if the number is divisible by 2 there is no point in checking to see if it is divisible by any larger even numbers. This suggests that the values used for `test-divisor` should not be 2, 3, 4, 5, 6, ..., but rather 2, 3, 5, 7, 9, To implement this change, define a procedure `next` that returns 3 if its input is equal to 2 and otherwise returns its input plus 2. Modify the `smallest-divisor` procedure to use `(next test-divisor)` instead of `(+ test-divisor 1)`. With `timed-prime-test` incorporating this

modified version of `smallest-divisor`, run the test for each of the 12 primes found in Exercise 1.22. Since this modification halves the number of test steps, you should expect it to run about twice as fast. Is this expectation confirmed? If not, what is the observed ratio of the speeds of the two algorithms, and how do you explain the fact that it is different from 2?

2.37.2 A Comedy of Error (just the one)

```

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

```

```

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

```

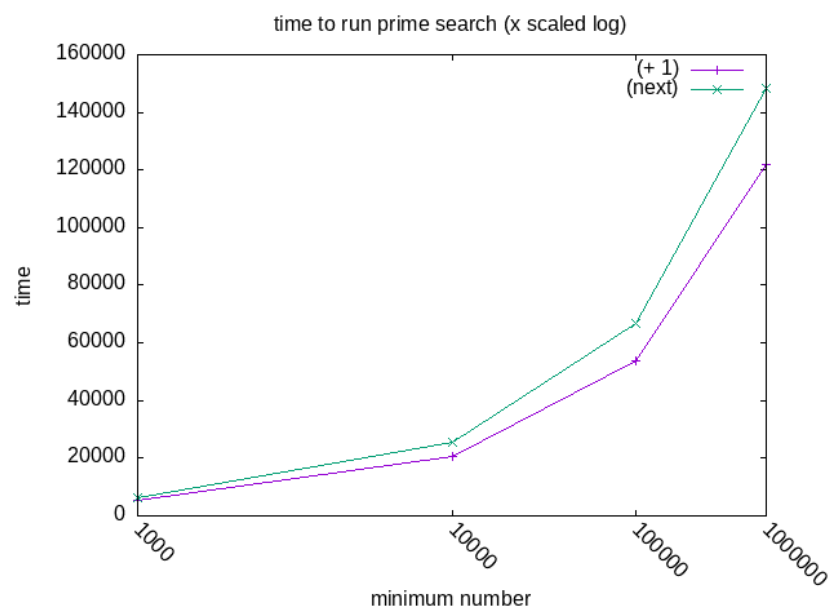
```

13      (cadr (mattbench2 (λ() (f 1000000 3))
14      ↪ benchmark-iterations))))
14
15      (print-row
16      (testit search-for-primes))

```

6456.538118 25550.757304 66746.041644 148505.580638

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



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

```

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

```


2.37.3 Answer

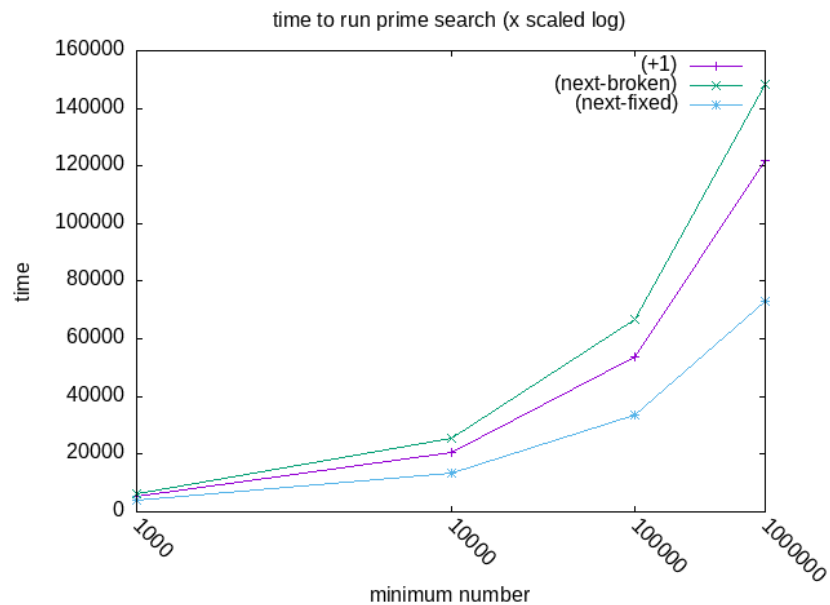
Ok, let's try that again.

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

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

3863.7424 13519.209814 33520.676384 73005.539932

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



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

Anyways, there's the speedup that was expected. Let's compare the ratios. Defining a new average that takes arbitrary numbers of arguments:

```
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) (* 100 (/ x y)))
```

```

6                                     (car smd) (car smdff)))
7                                     #nil))
8 (cons "% speedup for real (next)"
9       (cons (format #f "~2$"
10                (apply average
11                      (map (lambda (x y) (* 100 (/ x y)))
12                           (car smd) (car smdff)))))
13       #nil)))

```

```

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

```

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

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

```

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

```

```

1 <<mattbench2>>
2 <<find-divisor-faster-real2>>
3 (define (prime? n)
4   (= n (smallest-divisor n)))
5 <<search-for-primes-untimed>>

```

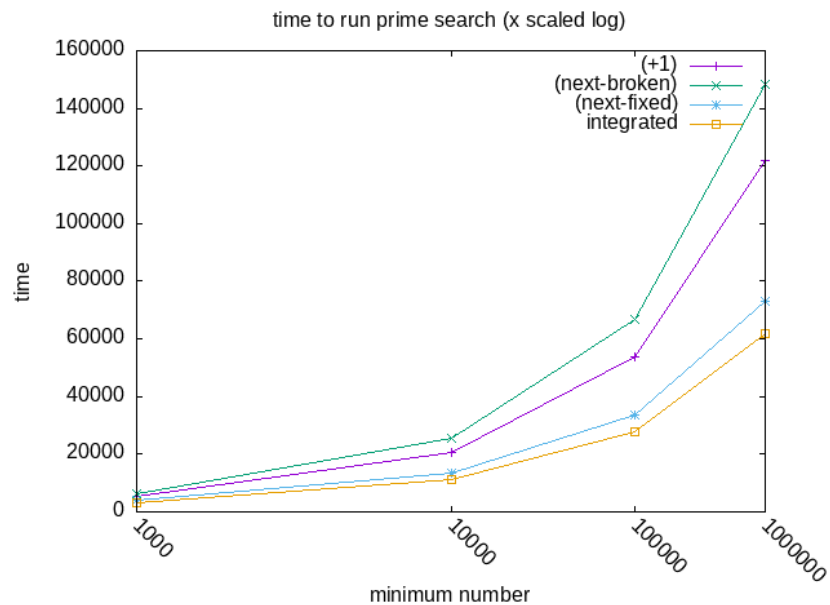
```

6 <<print-table>>
7
8 (define benchmark-iterations 500000)
9 (define (testit f)
10   (list (cadr (mattbench2 (λ() (f 1000 3)) benchmark-iterations))
11         (cadr (mattbench2 (λ() (f 10000 3)) benchmark-iterations))
12         (cadr (mattbench2 (λ() (f 100000 3))
13   ↪ benchmark-iterations))
14         (cadr (mattbench2 (λ() (f 1000000 3))
15   ↪ benchmark-iterations))))
16
17 (print-row
18   (testit search-for-primes))

```

3151.259574 11245.20428 27803.067944 61997.275154

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



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

2.38 Exercise 1.24

2.38.1 Text

```

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

2.38.3 Answer

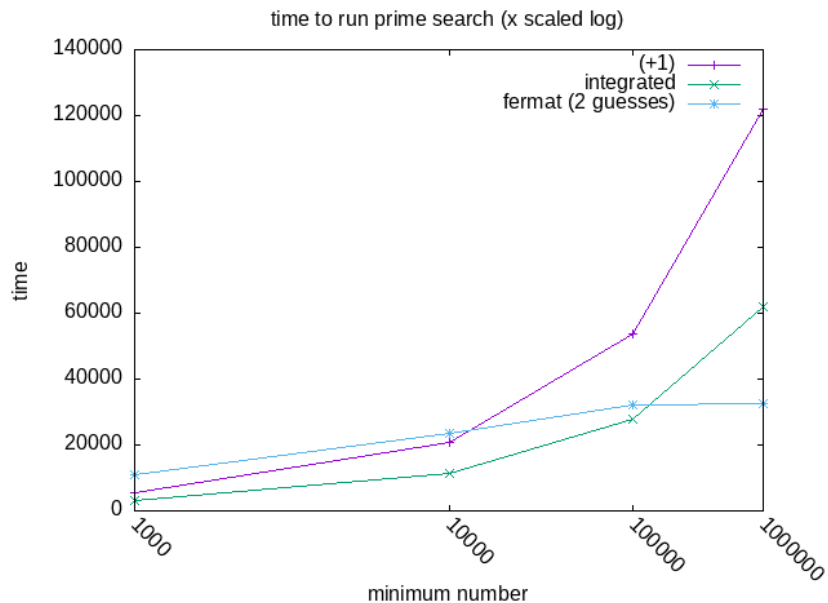
```

1  <<mattbench2>>
2  <<expmod>>
3  <<fermat-test>>
4  <<fast-prime>>
5  (define feramat-iterations 2)
6  (define (prime? n)
7    (fast-prime? n feramat-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)) benchmark-iterations))
14         (cadr (mattbench2 (λ() (f 10000 3)) benchmark-iterations))
15         (cadr (mattbench2 (λ() (f 100000 3))
16   ↪ benchmark-iterations))
17         (cadr (mattbench2 (λ() (f 1000000 3))
18   ↪ benchmark-iterations)))))
19 (print-row
  (testit search-for-primes))

```

11175.799722 23518.62116 32150.745642 32679.766448

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



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

```

1 <<mattbench2>>
2 <<find-divisor-faster-real>>
3 (define (prime? n)
4   (= n (smallest-divisor n)))
5 <<search-for-primes-untimed>>
6 <<print-table>>
7
8 (define benchmark-iterations 100000)
9 (define (testit f)
10   (list (cadr (mattbench2 (λ() (f 1000 3)) benchmark-iterations))
11         (cadr (mattbench2 (λ() (f 10000 3)) benchmark-iterations))
12         (cadr (mattbench2 (λ() (f 100000 3))
13   ↪ benchmark-iterations))
14         (cadr (mattbench2 (λ() (f 1000000 3))
15   ↪ benchmark-iterations))
16         (cadr (mattbench2 (λ() (f 10000000 3))
17   ↪ benchmark-iterations))
18         (cadr (mattbench2 (λ() (f 100000000 3))
19   ↪ benchmark-iterations))
20         (cadr (mattbench2 (λ() (f 1000000000 3))
21   ↪ benchmark-iterations))
22         (cadr (mattbench2 (λ() (f 10000000000 3))
23   ↪ benchmark-iterations)))

```

```

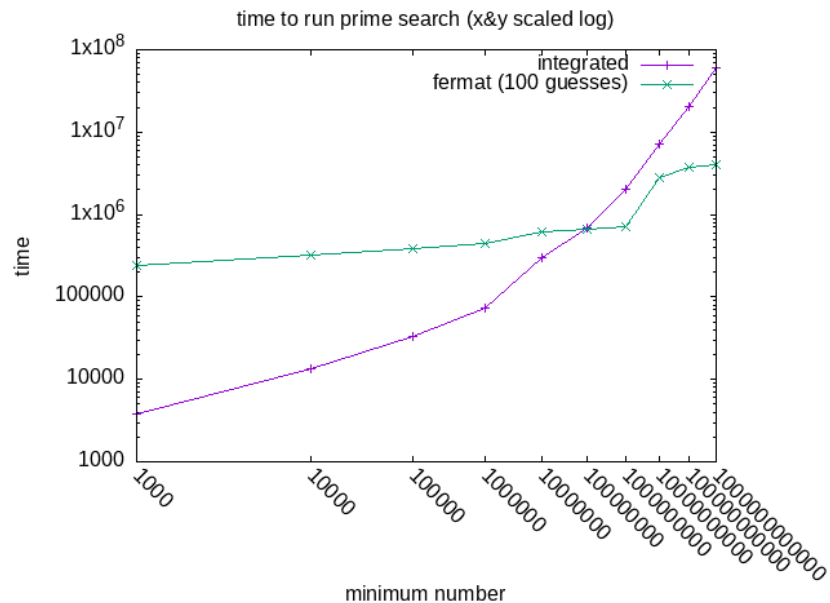
17      (cadr (mattbench2 (λ() (f 10000000000 3))
↳ benchmark-iterations))
18      (cadr (mattbench2 (λ() (f 100000000000 3))
↳ benchmark-iterations))
19      (cadr (mattbench2 (λ() (f 1000000000000 3))
↳ benchmark-iterations))))
20
21 (print-row
22 (testit search-for-primes))

```

```

1  <<mattbench2>>
2  <<expmod>>
3  <<fermat-test>>
4  <<fast-prime>>
5  (define fermat-iterations 100)
6  (define (prime? n)
7    (fast-prime? n fermat-iterations))
8  <<search-for-primes-untimed>>
9  <<print-table>>
10
11 (define benchmark-iterations 100000)
12 (define (testit f)
13   (list (cadr (mattbench2 (λ() (f 1000 3)) benchmark-iterations))
14         (cadr (mattbench2 (λ() (f 10000 3)) benchmark-iterations))
15         (cadr (mattbench2 (λ() (f 100000 3))
↳ benchmark-iterations))
16         (cadr (mattbench2 (λ() (f 1000000 3))
↳ benchmark-iterations))
17         (cadr (mattbench2 (λ() (f 10000000 3))
↳ benchmark-iterations))
18         (cadr (mattbench2 (λ() (f 100000000 3))
↳ benchmark-iterations))
19         (cadr (mattbench2 (λ() (f 1000000000 3))
↳ benchmark-iterations))
20         (cadr (mattbench2 (λ() (f 10000000000 3))
↳ benchmark-iterations))
21         (cadr (mattbench2 (λ() (f 100000000000 3))
↳ benchmark-iterations))
22         (cadr (mattbench2 (λ() (f 1000000000000 3))
↳ benchmark-iterations))))
23
24 (print-row
25 (testit search-for-primes))

```

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

2.39 Exercise 1.25

2.39.1 Question

Alyssa P. Hacker complains that we went to a lot of extra work in writing `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) ;; <== hmm.
6                (expmod base (/ exp 2) m))
7              m))
8          (else
9            (remainder
10              (* base
11                (expmod base (- exp 1) m))
12              m))))

```

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

2.40.2 Answer

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

2.41 Exercise 1.27

2.41.1 Question

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

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

2.42.2 Analysis

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

```

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

```

```

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

```

```

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

```

This will be useful in various future functions:

```

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

```

```

1 (use-srfis '(1))
2 <<expmod>>
3 <<fermat-prime?>>
4 <<find-divisor-faster-real>>
5 (define (prime? n)

```

```

6      (= n (smallest-divisor n)))
7      <<get-lists-of-primes>>
8      (define prime-is-working
9        (and (and-map prime? list-of-primes)
10             (not (and-map prime? list-of-carmichaels))))
11      (format #t "(prime?) is working: ~a~%"
12              (if prime-is-working
13                  "Yes"
14                  "No"))
15      (define fermat-is-vulnerable
16        (and (and-map fermat-prime? list-of-primes)
17             (and-map fermat-prime? list-of-carmichaels)))
18      (format #t "(fermat-prime?) is vulnerable: ~a~%"
19              (if fermat-is-vulnerable
20                  "Yes"
21                  "No"))

```

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

2.42.3 Answer

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

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

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

```

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

```

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

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

```

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

```

One more time.

```

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

```

```

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

```

```

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

```

2.43 1.3: Formulating Abstractions with Higher-Order Procedures

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

2.44 1.3.1: Procedures as Arguments

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

For the program:

```

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

```



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

Which is equivalent to:

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

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

2.45 Exercise 1.29

2.45.1 Text

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

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

2.45.2 Question

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

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

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

2.45.3 Answer

```

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

```

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

```

1 <<mattbench2>>
2 <<print-table>>
3 (define (cube x)
4   (* x x x))
5 <<sum>>
6 <<integral>>
7 <<int-simp>>
8
9 (define iterations 100000) ;; benchmark iterations
10 (define (run-test1)
11   (integral cube 0.0 1.0 0.0008))
12 (define (run-test2)
13   (int-simp cube 0.0 1.0 1000.0))
14 (print-table (list (list "integral dx:0.0008" "int-simp i:1000")
15                     (list (run-test1) (run-test2))
16                     (list (cadr (mattbench2 run-test1 iterations))
17                           (cadr (mattbench2 run-test2
18                               iterations)))))
19   #: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 0
```

2.47.3 Question A.2

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

2.47.4 Answer A.2

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

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

2.47.5 Question A.3

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

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

2.47.6 Answer A.3

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

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

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

3.1415769458228726

2.47.7 Question B

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

2.47.8 Answer B

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

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

iterative	recursive
1267118.0538	3067085.5323

2.48 Exercise 1.32

2.48.1 Question A

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

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

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

2.48.2 Answer A

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

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

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

5040

2.48.3 Question B

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

2.48.4 Answer B

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

2.49 Exercise 1.33

2.49.1 Question A

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

2.49.2 Answer A

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

2.49.3 Question B

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

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

```

1 (load "mattcheck.scm")
2 (define (square x)
3   (* x x))
4 <<filtered-accumulate-iter>>
5 <<expmod-mr2>>
6 <<mr-test2>>
7 <<mr-prime>>
8 (define mr-times 100)
9 (define (prime? x)
10   (mr-prime? x mr-times))
11 (define (prime-sum a b)
12   (filtered-accumulate-iter prime? + 0
13                               square a 1+ b))
14
15 (mattcheck-equal "1 prime correct"
16                 (prime-sum 1008 1010)
17                 (square 1009)) ;; 1009
18 (mattcheck-equal "many primes correct"
19                 (prime-sum 1000 2001)
20                 (apply +
21                         (map square
22                             (filter prime? (iota (- 2001
23   ↪ 1000)
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647

```



```

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 n))
19             (iota (- n 1) 1))))
20
21 (mattcheck-equal "Ex_1-33B"
22                  (Ex_1-33B 100)
23                  (alternate 100))

```

SUCCEED at Ex_1-33B

2.50 1.3.2: Constructing Procedures Using lambda

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

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

2.51 Exercise 1.34

2.51.1 Question

Suppose we define the procedure

```

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

```

Then, we have

```

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

```

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

2.51.2 Answer

It ends up trying to execute 2 as a function.

```

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

```

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

2.52 1.3.3 Procedures as General Methods

The **half-interval method**: if $f(a) < 0 < f(b)$, then f must have at least one 0 between a and b . To find 0, let x be the average of a and b , if $f(x) < 0$ then 0 must be between x and b , if $f(x) > 0$ 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 equivalent to `(lambda (x) (+ x x))`. In English, "the function whose value at y is x/y ". *Though it seems like \mapsto doesn't necessarily describe a function, but the value of a function at a certain point? Or maybe that would just be , ie $f(x)$ etc*

2.53 Exercise 1.35

2.53.1 Text

```

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

```

```

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

```

```

11      (try next)))
12  (try first-guess))

```

2.53.2 Question

Show that the golden ratio φ is a fixed point of the transformation $x \mapsto 1 + 1/x$, and use this fact to compute φ by means of the 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 "~8~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 (λ(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 (λ(x) (average (log x) (/ (log 1000) (log x)))) 1.1)

```

Damped, it makes 21.

2.55 Exercise 1.37

2.55.1 Question A

An infinite continued fraction is an expression of the form

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

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 φ is the golden ratio (described in 1.2.2). One way to approximate an infinite continued fraction is to truncate the expansion after a given number of terms. Such a truncation—a so-called k -term finite continued fraction—has the form

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

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) result)))))
6
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)

```

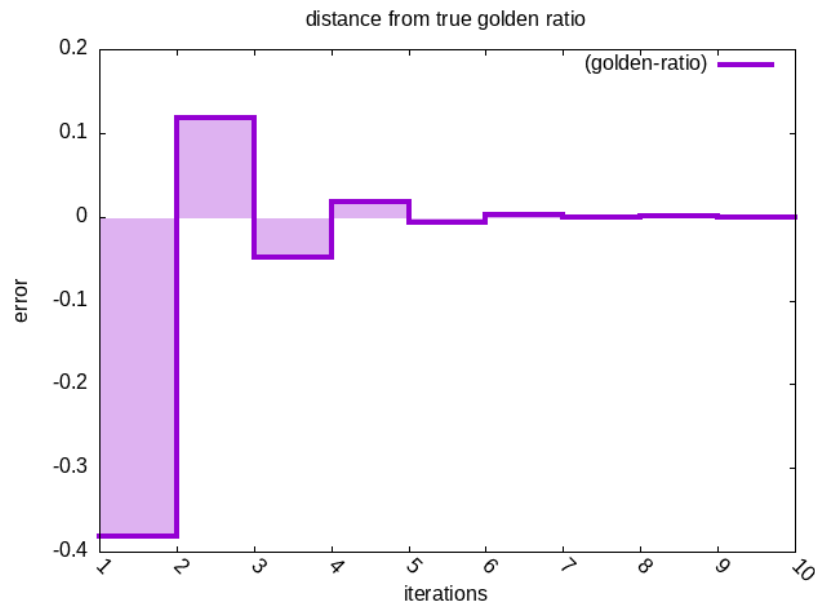
for successive values of `k`. How large must you make `k` in order to get an approximation that is accurate to 4 decimal places?

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
7   (rec 1))

```

```

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

```

```

7
8 (load "mattcheck.scm")
9 (mattcheck-equal "cont-frac iter and recursive equivalence"
10                 (golden-ratio-rec 15)
11                 (golden-ratio 15))

```

SUCCEED at cont-frac iter and recursive equivalence

2.56 Exercise 1.38

2.56.1 Question

In 1737, the Swiss mathematician Leonhard Euler published a memoir *De Fractionibus Continuis*, which included a continued fraction expansion for $e - 2$, where e is the base of the natural logarithms. In this fraction, the N_i are all 1, and the D_i are successively 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, Write a program that uses your `cont-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-fraction>>
2  (define (tan-cf x k)
3    (cont-frac (lambda (i) (if (= i 1)
4                               x
5                               (* x x -1.0))))
6    (lambda (i) (if (= i 1)
7                    1.0
8                    (- (* i 2.0) 1.0)))
9    k))
10
11 (tan-cf 55 101)

```

-45.1830879105221

2.58 1.3.4 Procedures as Returned Values

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

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

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

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

2.59 Exercise 1.40

2.59.1 Text

```

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

```

```

1  (define dx 0.00001)

```



```

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

```

```

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

```

```

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

```

2.59.2 Question

Define a procedure `cubic` that can be used together with the `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^{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^{th} repeated application of f is the function $x \mapsto x + n$. If f is the operation of squaring a number, then the n^{th} repeated application of f is the function that raises its argument to the 2^n -th power. Write a procedure that takes as inputs a procedure that computes f and a positive integer n and returns the procedure that computes the n^{th} repeated application of f .

2.62.2 Answer

```
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    (λ(x)
4      (average (f (- x dx))
5                (f x)
6                (f (+ x dx)))))
7  (define (smooth-n f n)
8    ((repeated smooth n) f))

```

2.64 Exercise 1.45

2.64.1 Question

We saw in 1.3.3 that attempting to compute square roots by naively finding a fixed point of $y \mapsto x/y$ does not converge, and that this can be fixed by average damping. The same method works for finding cube roots as fixed points of the average-damped $y \mapsto x/y^2$. Unfortunately, the process does not work for fourth roots—a single average damp is not enough to make a fixed-point search for $y \mapsto x/y^3$ converge. On the other hand, if we average damp twice (i.e., use the average damp of the average damp of $y \mapsto x/y^3$) the fixed-point search does converge. Do some experiments to determine how many average damps are required to compute n^{th} roots as a fixed-point search based upon repeated average damping of $y \mapsto x/y^{n-1}$. Use this to implement a simple procedure for computing n^{th} roots using `fixed-point`, `average-damp`, and the `repeated` procedure of Exercise 1.43. Assume that any arithmetic operations you need are available as primitives.

2.64.2 Answer

So this is strange. Back in my original workthrough of this book, I'd decided that finding an 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 asked to repeat
   ↳ zero times.
2  <<compose>>
3  <<identity>>
4  (define (repeated f n)
5    (define (rec m)
6      (if (= n 1)
7          f
8          (repeated (compose f f)
9                    (- n 1))))
10   (if (= n 0)
11       identity
12       (rec n)))

```

```

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

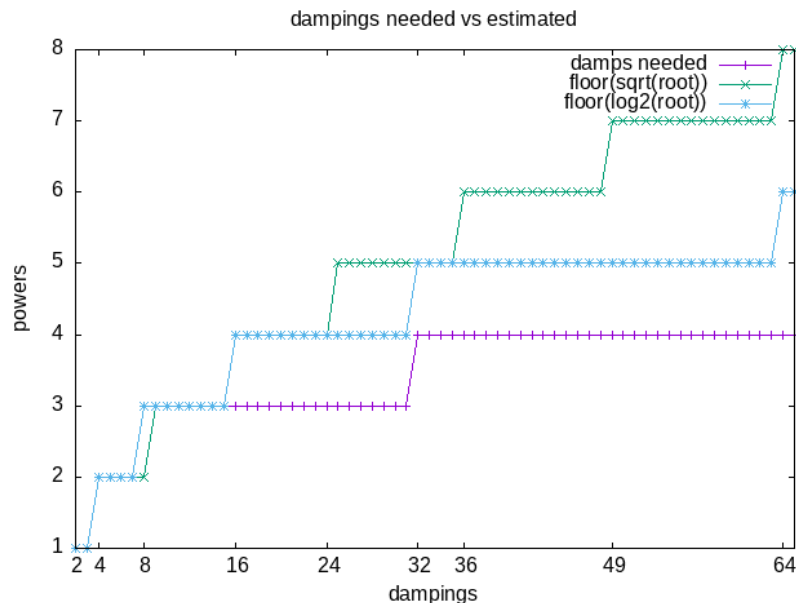
```

Let's automatically find how many dampings are necessary. We can make a program that finds higher and higher 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 damp)
18                     (lambda (y)
19                       (/ base (expt y (- n 1))))
20                     1.1)))
21            (if (string? x)
22                (try (1+ damp))
23                (list base n x damp))))))
24      (if (> n n-max)
25          ll
26          (iter (cons (try 1) ll))))))
27
28  (iter '())
29  (let* ((t (reverse (nth-tester 3 65))))
30    (cons '("root" "result" "damps needed" "floor(sqrt(root))"
31          ↪ "floor(log2(root))")
32          (map (λ(x)
33                (append x
34                        (list (floor (sqrt (car x)))
35                              (floor (/ (log (car x)) (log 2))))))
36              (map cdr t))))

```



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

2.65 Exercise 1.46

2.65.1 Question

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

2.65.2 Answer

```

1 (define (iterative-improve good-enough? improve)
2   (λ(first-guess)
3     (define (iter guess)
4       (let ((next (improve guess)))
5         (if (good-enough? guess next)
6             next
7             (iter next))))
8     (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) 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 x))) 1.0)
6   (fixed-point-improve (λ(x)(+ 1 (/ 1 x))) 1.0))
7
8 <<sqrt>>
9 <<sqrt-improve>>
10 (mattcheck "sqrt-improve still working"
11   (sqrt 5)
12   (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 (+ (* (numer x) (denom y))
3                (* (numer y) (denom x)))
4             (* (denom x) (denom y))))
5
6 (define (sub-rat x y)
7   (make-rat (- (* (numer x) (denom y))
8                (* (numer y) (denom x)))
9             (* (denom x) (denom y))))
10
11 (define (mul-rat x y)
12   (make-rat (* (numer x) (numer y))
13             (* (denom x) (denom y))))
14
15 (define (div-rat x y)
16   (make-rat (* (numer x) (denom y))
17             (* (denom x) (numer y))))
18
19 (define (equal-rat? x y)
20   (= (* (numer x) (denom y))
```

```
21 (* (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 seg))
16                        (x-point (end-segment seg)))
17              (average (y-point (start-segment seg))
18                        (y-point (end-segment seg)))))
19 (define (midpoint-segment-opt seg)
```

```

20 (let ((ax (x-point (start-segment seg)))
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))))
63

```

```

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

```

```

45 (add-points
46   (rect-origin R)
47   (add-points
48     (vector-to-xy (rect-width R) (rect-angle R))
49     (vector-to-xy (rect-height R)
50                   (+ 90 (rect-angle R))))))

```

```

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

```

```

38         (rect-area ABC)
39         (rect-area CDA)
40         1)
41     (mattcheck "rect-area WXYZ"
42       (rect-area WXY)
43       3.0)
44     (mattcheck "rect-peri ABCD"
45       (rect-peri ABC)
46       4)
47     (mattcheck "rect-peri WXYZ"
48       (rect-peri WXY)
49       7.0710678118654755))

```

```

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

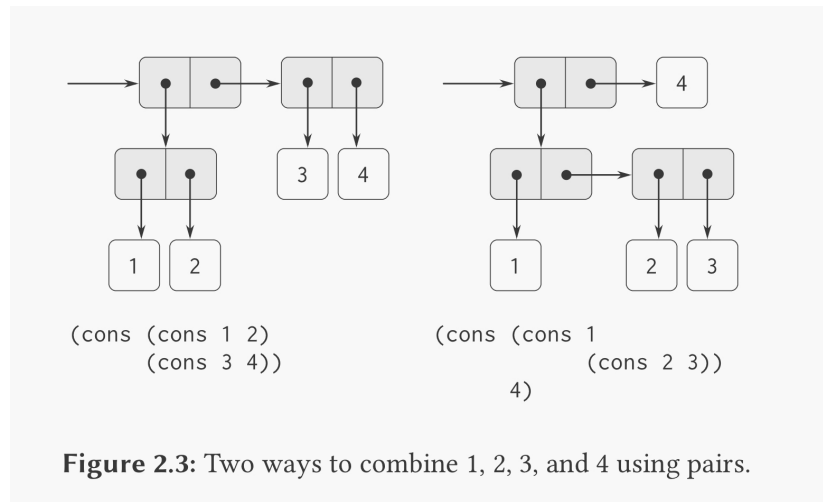
```

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 2.2: Hierarchical Data and the Closure Property

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



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.

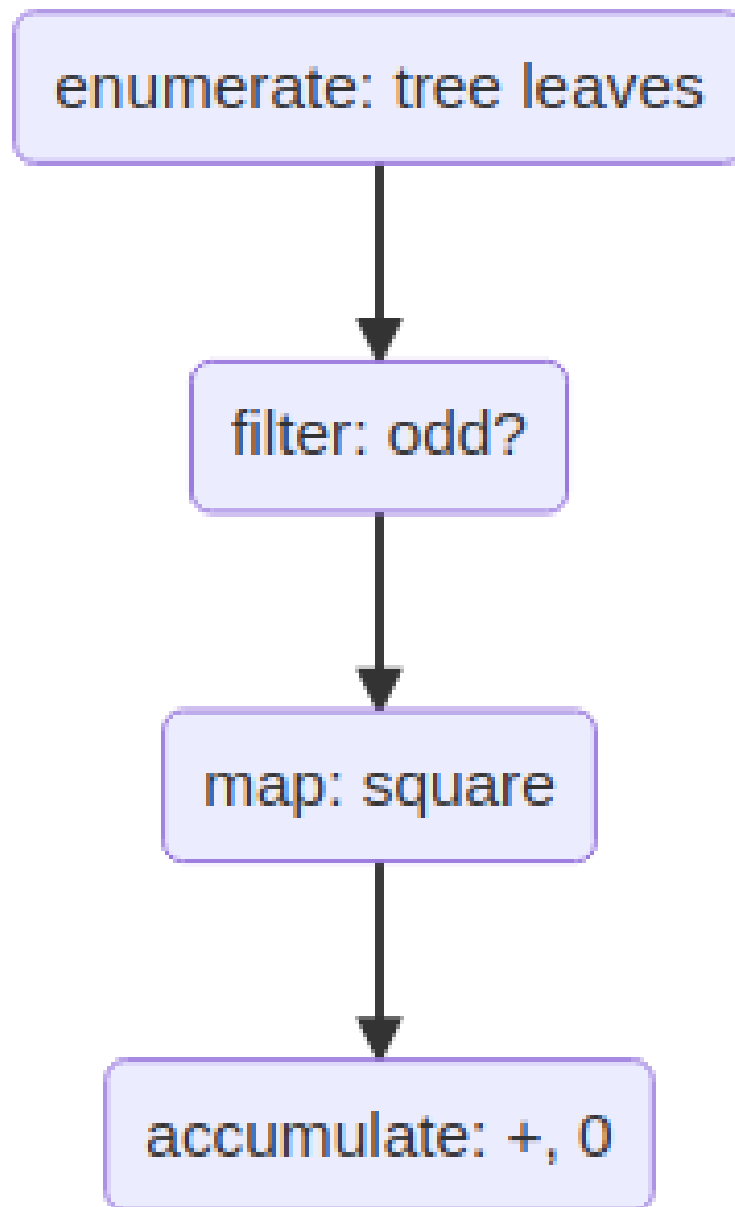
Definitions of “closure”

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.

3.7 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:



3.8 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.