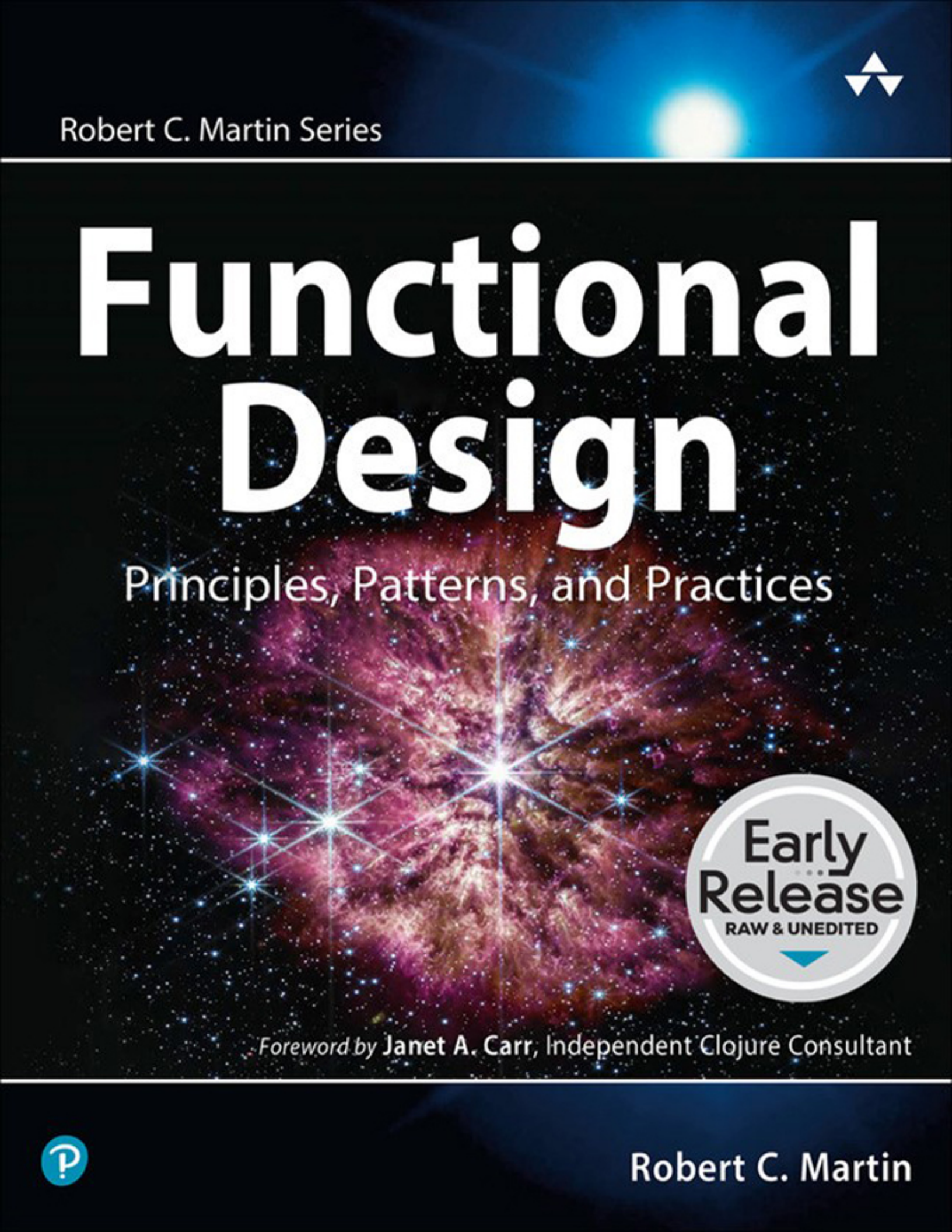
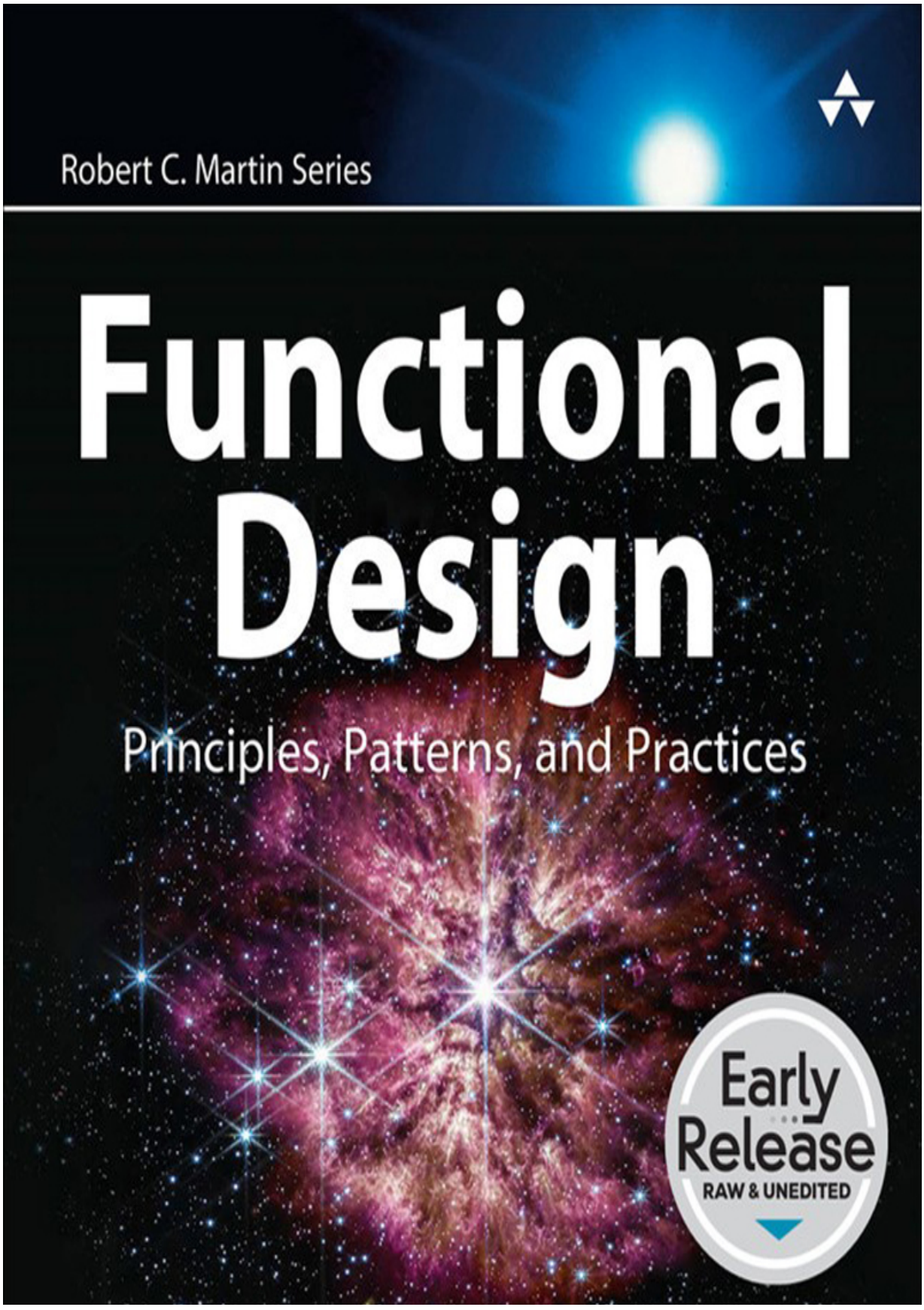

Robert C. Martin Series
Functional Design
Principles, Patterns, and Practices
Early Release
RAW & UNEDITED
Foreword by Janet A. Carr, Independent Clojure Consultant
Robert C. Martin


Robert C. Martin Series
Functional Design
Principles, Patterns, and Practices
Early Release
RAW & UNEDITED


Foreword by Janet A. Carr, Independent Clojure Consultant
Robert C. Martin

Functional Design: Principles, Patterns, and Practices

Robert C. Martin

Addison-Wesley Professional

Contents

Foreword

Preface

Introduction

About the Author

I: Functional Basics

1. Immutability

2. Persistent Data

3. Recursion and Iteration

4. Laziness

5. Statefulness

II: Comparative Analysis

6. Prime Factors

7. Bowling Game

8. Gossiping Bus Drivers

9. Object Oriented Programming

10. Types

III: Functional Design

11. Data Flow

12. SOLID

IV: Functional Pragmatics

13. Tests

14. GUI

15. Concurrency

V: Design Patterns

16. Design Patterns

VI: Case Study

17. Case Study: Wator

Afterword

Table of Contents

Foreword

Preface

Introduction

About the Author

I: Functional Basics

1. Immutability

    What Is Functional Programming?

    The Problem with Assignment

    So Why Is It Called Functional?

    No Change of State?

    Immutability

2. Persistent Data

    On Cheating

    Making Copies

    Structural Sharing

3. Recursion and Iteration

    Iter ation

    Very Brief Clojure Tutorial

    Iteration

    TCO, Clojure, and the JVM

    Recursion

4. Laziness

- Lazy Accumulation
- OK, but Why?
- Coda

5. Statefulness
- When We MUST Mutate
- Software Transactional Memory (STM)
- Life Is Hard, Software Is Harder

II: Comparative Analysis

6. Prime Factors
7. Bowling Game
8. Gossiping Bus Drivers
9. Object Oriented Programming
10. Types

III: Functional Design

11. Data Flow
12. SOLID

IV: Functional Pragmatics

13. Tests
14. GUI
15. Concurrency

V: Design Patterns

16. Design Patterns

VI: Case Study

17. Case Study: Wator

Afterword

Foreword [This content is currently in development.]

This content is currently in development.

Preface [This content is currently in development.]

This content is currently in development.

Introduction [This content is currently in development.]

This content is currently in development.

About the Author [This content is currently in development.]

This content is currently in development.

I: Functional Basics

1. Immutability

What Is Functional Programming?

If you were to ask the average programmer what functional programming is, you might get any of the following answers.

• Programming with functions.

• Functions are “first class” elements.

• Programming with referential transparency.

• A programming style based upon lambda calculus.

While these assertions might be true, they are not particularly helpful. I think a better answer is: *Programming without assignment statements*.

Perhaps you don’t think that definition is much better. Perhaps it even frightens you. After all, what do assignment statements have to do with functions; and how can you possibly program without them?

Good questions. Those are the questions that I intend to answer in this chapter.

Consider the following simple C program:

```
int main(int ac, char** av) {
    while(!done())
        doSomething();
}
```

This program is the core loop of virtually every program ever written. It quite literally says: “Do something until you are done.” What’s more, this program has no visible assignment statements. It is functional? And if so, does that mean every program ever written is functional?

Let’s actually make this function do something. Let’s have it compute the sum of the squares of the first ten integers $[1..10]$:

```
int n=1;
int sum=0;
int done() {
  return n>10;
}
void doSomething() {
  sum += n*n;

  ++n;
}
void sumFirstTenSquares() {
    while(!done())

        doSomething();
}
```

This program is not functional because it uses two assignment statements in the `doSomething function`. It’s also just plain ugly with

those two global variables. Let's improve it:

```
int sumFirstTenSquares() {
  int sum = 0;
  int i=1;
loop:
  if (i>10)
    return sum;
  sum += i*i;
  i++;
  goto loop;
}
```

This is better; the two globals have become local variables. But it's still not functional. Perhaps you are worried about that `goto`. It is there for a good reason. Bear with me as you consider this small modification that uses a worker function to convert the local variables into function arguments:

```
int sumFirstTenSquaresHelper(int sum, int i) {
loop:
  if (i>10)
    return sum;
  sum += i*i;
  i++;
  goto loop;
}
```

```
int sumFirstTenSquares() {
  return sumFirstTenSquaresHelper(0, 1);
}
```

This program is still not functional; but it's an important *milestone* that we'll refer to in a moment. But now, with one last change, something magical happens:

```
int sumFirstTenSquaresHelper(int sum, int i) {
  if (i>10)
    return sum;
  return sumFirstTenSquaresHelper(sum+i*i, i+1);
}
int sumFirstTenSquares() {
  return sumFirstTenSquaresHelper(0, 1);
}
```

All the assignment statements are gone, and this program is functional. It's also recursive. That's no accident. If you want to get rid of assignment statements, you *have* to use recursion. Recursion allows you to replace the assignment of local variables with the *initialization* of function arguments.

It also burns up a lot of space on the stack. However, there is a little trick we can use to fix that problem.

Notice that the last call to `sumFirstTenSquaresHelper` is also the last use of `sum` and `i` in that function. Holding those two variables on the stack after initializing the two arguments of the recursive call is pointless; they'll never be used. What if, instead of creating a new stack frame for the recursive call, we simply reused the current stack frame by jumping back to the top of the function with a `goto`, as we did in the *milestone* program?

This cute little trick is called *tail call optimization* (TCO) and all functional languages make use of it.[1]

---

1. In one way or another. The JVM complicates TCO a bit. C, of course, does not do TCO, and so all my recursive examples in C will grow the stack.

Notice that TCO effectively turns that last program into the *milestone* program. The last three lines of `sumFirstTenSquaresHelper` in the *milestone* program are, in effect, the recursive function call. Does that mean that the *milestone* program is functional too? No, it just behaves identically. At the source code level, that program is not functional because it has assignment statements. But if we take one step back and ignore the fact that the local variables changed as opposed to being reinstantiated in a new stack frame, then the program *behaves* as a functional program.

As we will discover in the next section, that is not a distinction without a difference. In the meantime, just remember that when you use recursion to eliminate assignment statements, you are not necessarily wasting lots of space on the stack. The language you are using is almost certainly using TCO.

The Problem with Assignment

First let's define what we mean by assignment. Assigning a value to a variable *changes* the original value of the variable to the newly assigned value. It is the change that makes it assignment.

In C we initialize a variable this way:

```
int x = 0;
```

But we assign a variable this way:

```
x=1;
```

In the first case, the variable `x` comes into existence with the value `0`; prior to the initialization, there was no variable `x`. In the second case, the value of `x` is changed to `1`. This may not seem significant, but the implications are profound.

In the first case, we do not know if `x` is actually a variable. It could be a constant. In the second case, there is no doubt. We are varying `x` by assigning it a new value. Thus, we can say that functional programming is programming *without variables*. The values in functional programs *do not vary*.

Why is this desirable? Consider the following:

```
.
//Block A
.
x=1;
.
//Block B
.
```

The *state of the system* during the execution of `Block A` is different from the state of the system in `Block B`. This means that `Block A` must execute *before* `Block B`. If the position of the two blocks were swapped, the system would likely not execute correctly.

This is called a *sequential or temporal coupling*—a coupling in time; and it is something you are probably quite familiar with. `Open` must be called before `close`. `New` must be called before `delete`. `Malloc` must be called before `free`. The list of pairs[2] like this is endless. And in many ways, they are a bane of our existence.

2. They are like the Sith; always two there are.

How many times have you forgotten to close a file, or release a block of memory, or close a graphics context, or release a semaphore? How many times have you debugged a pernicious problem only to find that you can fix it by swapping the position of two function calls?

And then there's garbage collection.

Garbage collection is a horrible[3] hack that we have accepted into our languages because we are just so bad at managing temporal couplings. If we were adept at keeping track of allocated memory, we would not depend on some nasty background process to clean up after us. But the sad fact is that we are so truly terrible at managing temporal couplings that we celebrate the crutches we build to protect ourselves from them.

3. And, no, reference counting isn't any better.

And that doesn't take into account multiple threads. When two or more threads are competing for the processor, keeping the temporal couplings in the correct order becomes a much more significant challenge. Those threads may get the order correct 99.99 percent of the time; but every once in a great while they may execute in the wrong

order and cause all manner of mayhem. We call those situations *race conditions*.

Temporal couplings and race conditions are the natural consequence of programming with variables—of using assignment. Without assignment, there are no temporal couplings and there are no race conditions.[4] You cannot have a concurrent update problem if you never update anything. You cannot have an ordering issue within a function if the system state never changes within that function.

---

4. We'll see later that this is not entirely correct. As Spock was fond of saying: "There are always possibilities."

But perhaps it's time for a simple example. Here's our nonfunctional algorithm again; this time without the `goto`:

```
1: int sumFirstTenSquaresHelper(int sum, int i) {
2:   while (i<=10) {
3:     sum += i*i;
4:     i++;
5:   }
6:   return sum;
7: }
```

Now let's say you'd like to log the progress of the algorithm with a statement like this:

```
log("i=%d, sum=%d", i, sum);
```

Where would you put that line? There are three possibilities. If you add the `log` statement after line 2 or 4, then the logged data will be correct, and the difference will simply be whether you are logging before or after the computation. If you insert the `log` statement after line 3, then the logged data will be incorrect. That is a temporal coupling—an ordering problem.

Now consider our functional solution, with one interesting cosmetic change:

```
int sumFirstTenSquaresHelper(int sum, int i) {
  return (i>10) ? sum : sumFirstTenSquaresHelper(
}
```

There is only one place we can put our `log` statement, and it will log correct data.

So Why Is It Called Functional?

A function is a mathematical object that maps inputs to outputs. Given $y = f(x)$, there is a value of y for every value of x. Nothing else matters to f. If you give x to f, you will get y every single time. The state of the system in which f executes is irrelevant to f.

Or, to say that a different way, there are no temporal couplings with f. There is no special order in which f must be invoked. If you call f with x, you will get y no matter what else may have changed.

Functional programs are true functions in this mathematical sense. If you decompose a functional program into many smaller functions, each of those will also be a true function in the same mathematical sense. This is called *referential transparency*.

A function is referentially transparent if you can always replace the function call with its value. Let's try that with our functional algorithm for calculating the sum of the squares of the first ten integers:

```
int sumFirstTenSquaresHelper(int sum, int i) {
  return (i>10) ? sum : sumFirstTenSquaresHelper(
}
int sumFirstTenSquares() {
  return sumFirstTenSquaresHelper(0, 1);
}
```

When we replace the first call to `sumFirstTenSquaresHelper` with its implementation, it becomes:

```
int sumFirstTenSquares() {
  return (1>10) ? 0 : sumFirstTenSquaresHelper(0-
}
```

When we replace the next function call, it becomes:

```
int sumFirstTenSquares() {
  return
    (1>10) ? 0 :
      (2>10) ? 0+1*1
             : sumFirstTenSquaresHelper((0+1*1)-
                                        (1+1)+1)
}
```

I think you can see where this is going. Each call to `sumFirstTen-Squares Helper` simply gets replaced with its implementation with the arguments properly replaced.

Notice that you cannot do this simple replacement with the nonfunctional version of the program. Oh, you can unwind the loop if you like; but that's not the same as simply replacing each function call with its implementation.

So, functional programs are composed of true mathematical, referentially transparent functions. And that's why this is called functional programming.

No Change of State?

If there are no variables in functional programs, then functional programs cannot change state. How can we expect a program to be useful if it cannot change state?

The answer is that functional programs compute a new state from an old state, *without changing the old state*. If this sounds confusing, then the following example should clear it up:

```
State system(State s) {
  return isFinal(s) ? s : system(s);
}
```

You can start the `system` in some initial `state`, and it will successively move the `system` from `state` to `state` until the final `state` is reached. The `system` does not change a state variable. Instead, at each iteration, a new `state` is created from the old `state`.

If we turn TCO off and allow the stack to grow with each recursive call, then the stack will contain all the previous states, unchanged.

Moreover, the `system` functions as a true function in the mathematical sense. If you call `system` with `state1`, it will return `state2` every single time.

If you look closely at our functional version of `sumFirstTenSquares`, you will see that it uses precisely this approach to the changing of state. There are no variables, and no internal state. Rather, the algorithm moves from the initial state to the final state, one state change at a time.

Of course, our `system` function does not appear to be able to respond to any inputs. It simply starts at some initial `state` and then runs to completion. But with a simple modification we can create a "functional" program that responds quite nicely to input events:

```
State system(State state, Event event) {
  return done(state) ? state : system(state, getE
}
```

Now, the computed next `state` of the `system` is a function of the current `state` and an incoming `event`. And voila! We have created a very traditional finite state machine that can react to events in real time.

Notice the quotes I put around the word *functional* above. That is because `getEvent` is not referentially transparent. Every time you call it you will get a different result. Thus, you cannot replace the call with its return value. Does this mean that our program is not actually functional?

Strictly speaking, any program that takes input in this manner cannot be purely functional. But this is not a book about purely functional programs. This is a book about functional *programming*. The style of the program above is "functional," even if the input is not pure; and it is that style that we are interested in here.

So, here, for your entertainment, is a simple little real-time finite state machine that is written in C and is "functional." It is the time-honored subway turnstile example. Have fun with it.

```
#include <stdio.h>
typedef enum {locked, unlocked, done} State;
typedef enum {coin, pass, quit} Event;
void lock() {
  printf("Locking.\n");
}
void unlock() {
  printf("Unlocking.\n");
}
void thankyou() {
  printf("Thanking.\n");
```

```
}
void alarm() {
  printf("Alarming.\n");
}

Event getEvent() {
  while (1) {

    int c = getchar();
    switch (c) {

      case 'c': return coin;
      case 'p': return pass;
      case 'q': return quit;
    }
  }
}
State turnstileFSM(State s, Event e) {
  switch (s) {

    case locked:
    switch (e) {
      case coin:
      unlock();
      return unlocked;

      case pass:
      alarm();
      return locked;
```

```
      return locked;

      case quit:
      return done;
    }

    case unlocked:
    switch (e) {
      case coin:
      thankyou();
      return unlocked;


      case pass:
      lock();
      return locked;

      case quit:
      return done;
    }
    case done:
    return done;
  }
}
State turnstileSystem(State s) {
  return (s==done)? 0

                  : turnstileSystem(
                      turnstileFSM(s, getEvent()))
}
```

```
}
int main(int ac, char** av) {
  turnstileSystem(locked);

  return 0;
}
```

Keep in mind that C does not use TCO, and so the stack will grow until it is exhausted—though that may require quite a few operations in this case.

Immutability

What all this means is that functional programs contain no variables. Nothing in a functional program changes state. State changes are passed from one invocation of a recursive function to the next, without altering any of the previous states. If those previous states aren't needed, TCO can optimize them away; but in spirit they all still exist, unchanged, somewhere in a past stack frame.

If there are no variables in a functional program, then the values we name are all *constants*. Once initialized, those constants never go away, and never change. In spirit, the entire history of every one of those constants remains intact, unchanged, and immutable.

2. Persistent Data

So far this has seemed relatively simple. Programs written in the “functional” style are simply programs that have no variables. Rather than reassign values to variables, we use recursion to initialize new function arguments with new values. Simple.

But data elements are seldom as simple as we have so far imagined them to be. So let's take a look at a slightly more complicated problem, *The Sieve of Eratosthenes*:

```
package sieve;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
public class Sieve {
  boolean[] isComposite;

  static List<Integer> primesUpTo(int upTo) {
    return (new Sieve(upTo).getPrimes());
  }
  private Sieve(int upTo) {
    if (upTo < 1)
      upTo = 1;
    isComposite = new boolean[upTo+1];
    Arrays.fill(isComposite, false);
    isComposite[0] = isComposite[1] = true;
    for (int i=0; i< isComposite.length; i++)
      if (!isComposite[i])
        for (int c = i+i; c<isComposite.length; c

          isComposite[c] = true;
  }
  public List<Integer> getPrimes() {
    ArrayList<Integer> primes = new ArrayList<>()
```

```
      for (int i=0; i<isComposite.length; i++)
        if (!isComposite[i])
          primes.add(i);
      return primes;
    }
  }
```

This cute little Java program computes the prime numbers up to a limit. Notice all the assignment statements. There are variables everywhere, so this program must not be functional.

But then again, look at the static function at the top. `Sieve.primesUpTo` is a true mathematical function. Every time you call it with `n`, it will return the prime numbers up to `n`. So we can cheat and say that despite the fact that the underlying algorithm uses variables, the result of that algorithm is functional.

On Cheating

Our computers are, in some sense, finite *Turing machines*; they are not based upon lambda calculus. The Church–Turing thesis tell us that Turing machines and lambda calculus are equivalent forms; but that doesn't mean you can easily translate from one to the other. A functional program is a program that *looks like* lambda calculus but is implemented in a finite Turing machine. And that implementation requires that we cheat.

The first cheat we saw was TCO. We waved it away with an argument about pragmatics. After all, since we were never going to need all those historical stack frames, why should we keep them? But that's still a cheat. Under the hood, our implementation was changing the values of existing variables. From the Turing machine's point of view, all our supposed constants were actually variables.

We could continue to push that cheat upward. This lovely little `Sieve` algorithm runs entirely in the constructor, so it's all *initialization*! And as we learned, initialization is not assignment. So the fact that this program has variables under the hood is no different from TCO. In the end, the result is still functional.

This is fun! We can keep pushing that cheat upward. We can push it up until it is outside our finite Turing machine of a computer. And then we could say to ourselves: "Every program that runs in this computer is functional because it will always produce the same outputs when given the same inputs. Never mind that the inputs and outputs include every single bit in the computer's memory. Never mind that. Yeah. That's the ticket."

Of course, if we take that view, then there's not much point in studying functional programming, is there? So let's back down from that highest-level cheat and keep pushing the cheats back down until we simply cannot practically escape them.

There is no reasonable escape from TCO. We don't have an infinite stack. We don't want our functional programs uselessly consuming gigabytes of stack space until they crash. So TCO is a practically unavoidable cheat.

Making Copies

So, what about that `Sieve` algorithm: Can we push the cheating down lower than that? Can we write that algorithm so that it does not use any assignment statements?

The problem, of course, is all those `for` loops. We need to turn those into recursive functions in order to get rid of the assignment statements. We also need to do something about the two arrays. We can't be changing elements in existing arrays, can we? That would make those arrays variables. So we'll have to make copies of them whenever we need to change an element:

```
package sieve;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
public class Sieve {
  static List<Integer> primesUpTo(int upTo) {
    return getPrimes(
      computeSieve(
        makeSieve(Math.max(upTo, 1)),
```

```
        makeSieve(nthPrime(upTo) + 1)),
        0),
      new ArrayList<>(), 0);
  }
  private static boolean[] makeSieve(int upTo) {
    boolean[] sieve = new boolean[upTo + 1];
    Arrays.fill(sieve, false);
    sieve[0] = sieve[1] = true;
    return sieve;

  }
  private static boolean[] computeSieve(boolean[]
    if (n >= sieve.length)
      return sieve;
    else if (!sieve[n])
      return computeSieve(markMultiples(sieve, n,
    else return computeSieve(sieve, n + 1);
  }
  private static boolean[] markMultiples(boolean[
                                         int prim
                                         int m) {
    int multiple = prime * m;
    if (multiple >= sieve.length)
      return sieve;
    else {
      var markedSieve = Arrays.copyOf(sieve, siev
      markedSieve[multiple] = true;
      return markMultiples(markedSieve, prime, m
    }
  }
```

```
    }
    public static List<Integer> getPrimes(boolean[]
                                          List<Inte
                                          int n) {
      if (n >= sieve.length)
        return primes;
      else if (!sieve[n]) {
        var newPrimes = new ArrayList<>(primes);
        newPrimes.add(n);

        return getPrimes(sieve, newPrimes, n + 1);
      } else {
        return getPrimes(sieve, primes, n + 1);
      }
    }
  }
```

That's not very pretty, is it. It is, however, pretty functional. You might complain about the assignments in `makeSieve`, and I agree that's a bit of a cheat, but it looks close enough to an initialization to satisfy me.

So, yes, all the significant assignment operations have been eliminated. All the named entities are constants, and the stack (if not deleted by TCO) contains the history of each invocation of each recursive function.

But at what cost? Every time either of the two arrays is modified, a new array is created in order to prevent the previous one from being changed. The amount of memory used by this algorithm could be enormous. Imagine finding all the primes up to $100{,}000$. How many `sieve` arrays would be created? How many `primes` arrays?

And what about execution time? Copying all those arrays over and over again must eat up a terrifying number of cycles.

Is that, then, the cost of functional programming? Must we live with such a huge extravagance of memory and time?

Structural Sharing

Fortunately, no. It turns out that there are data structures that behave very much like arrays but that also efficiently maintain the history of their past states. These data structures are *n*-ary trees. The bigger the *n*, the more efficient they are. But for the sake of simplicity, I will choose an *n* of 2—binary trees—for the following examples.

Let us say that we wish to represent a simple array of integers from 1 to 8. The binary tree that achieves this is shown in Figure 2.1.

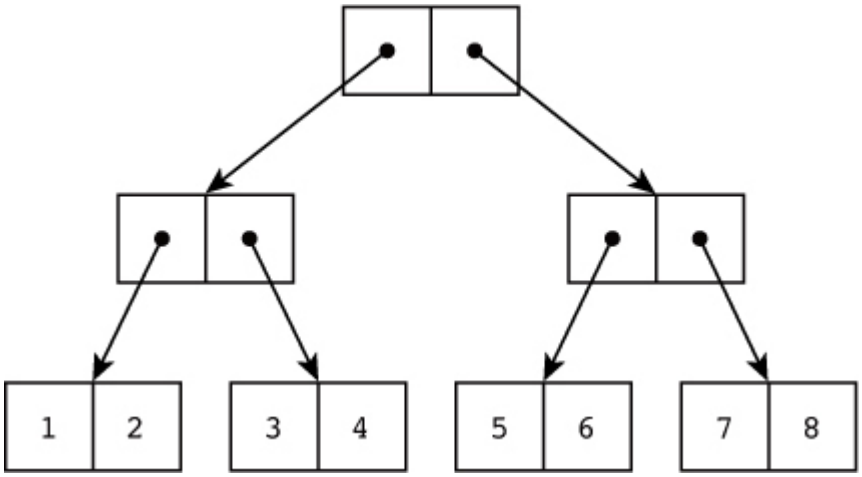


Figure 2.1 A binary tree representing an array of integers [1..8]

If you look at the leaves and ignore the branches, you will see that the leaves form an array. The branches simply provide a way to traverse to each leaf in some ordered way. That order is the index of the array!

To get to the element at index 0 of the array, simply take the leftmost branch of each node. To get to the element at index 1, go left at each node but right at the last node.

I won't belabor this point. I'm sure you all understand binary trees.

Now, let's say we want to append a 42 on the end of this array while preserving the existence of the previous array. The binary tree that achieves this is shown in Figure 2.2.

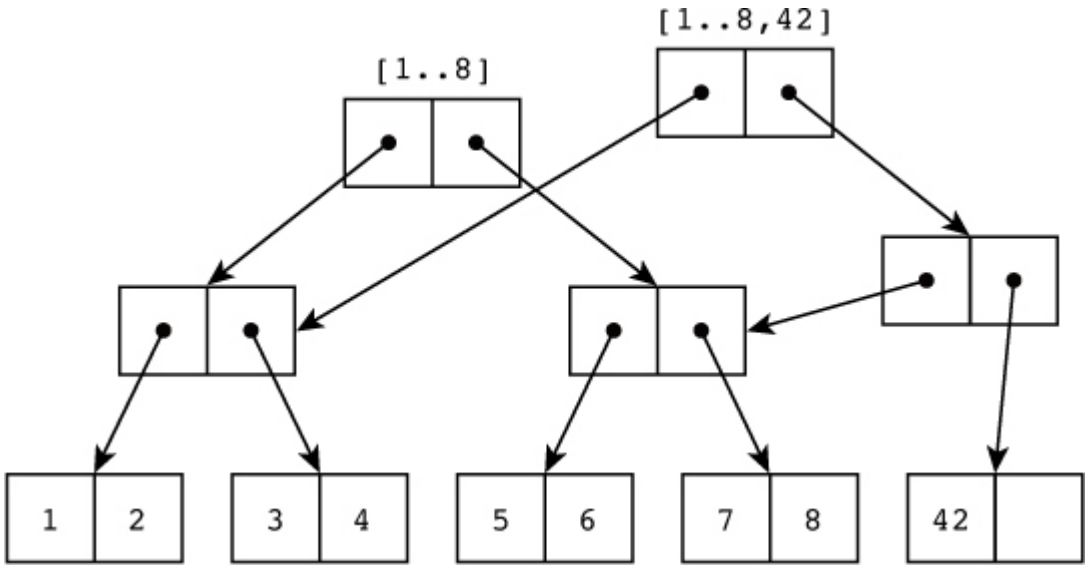


Figure 2.2 A binary tree that represents [1..8, 42] but also preserves the original [1..8] array

Now the tree has *two roots*. The root at the top left still represents the array from 1..8. The root at the top right represents the new array with a 42 appended after the 8.

Stop now and think carefully about this. It should be clear that representing linear arrays as trees, in the manner shown, will allow us to represent additions, insertions, and deletions while preserving all previous arrangements, without massive copying of the array.

Oh, there is some copying going on. We may have to copy a leaf node, or some of the branch nodes, depending on what operation we are performing. But the amount of memory and the number of cycles are drastically less than simply maintaining copies of all the past versions of the array.

In the end, every past version of the array will be represented by a new root node connected to a small number of additional branch nodes, allowing the majority of the elements of the array to be shared among all the versions.

Now consider what happens if we use 32-ary trees instead of binary trees. For arrays of a million elements, the tree depth is on the order of four or five branches. Copying five nodes of 32 elements each is *a lot* faster and requires *a lot* less memory than copying a million elements. Indeed, the cost, while not zero, is so small as to be inconsequential for most applications.

So we have a way to represent an indexable linear array that can be versioned over time while preserving all past versions. We call this *persistence*.[1] A persistent data structure has the ability to undergo change while remembering all past versions of itself.

---

1. Not to be confused with the overloaded term used to describe data in offline storage.

But what about higher-level data structures like hash maps, sets, stacks, and queues? How do we make all of them as persistent as our linear indexed array? Of course, all those data structures can be implemented using indexed arrays. Indeed, since the memory of the computer is nothing more than one big indexed linear array, every

data structure that you can represent within a computer can also be represented in a persistent array.

And so the problem we confronted at the start of this chapter, the problem of copying, can be set aside. The cost of functional programming, in memory and cycles, need not dissuade us from further study and pursuit of the benefits of functional programming.

And with that problem solved, all future examples will be written in Clojure, a language that intrinsically supports persistent data structures.

3. Recursion and Iteration

In Chapter 1, Immutability, I stated that functional programming makes use of recursion in order to eliminate assignment. In this chapter, we will look at the two different varieties of recursion; one we will call iteration and the other will retain the original name: recursion.

Iter ation

TCO is the remedy for the infinite stack depth implied by infinite recursive loops. However, TCO is only applicable if the recursive call is the very last thing to be executed within the function. Such functions are often called *tail call functions*.

Here is a very traditional implementation of a function to create a list of Fibonacci numbers:

```
(defn fibs-work [n i fs]
  (if (= i n)
    fs
    (fibs-work n (inc i) (conj fs (apply + (take-
(defn fibs [n]
  (cond
    (< n 1) []
    (= n 1) [1]
    :else (fibs-work n 2 [1 1])))
```

This program is written in Clojure, which is a variant of Lisp. You call this function like this:

```
(fibs 15)
```

And it returns an array of the first 15 Fibonacci numbers:

```
[1 1 2 3 5 8 13 21 34 55 89 144 233 377 610]
```

Many programmers experience eyestrain headaches the first few times they look at Lisp, mostly because the parentheses don't seem to make any sense. So let me give you a very brief tutorial about those parentheses.

Very Brief Clojure Tutorial

1. This is a typical function call in C, C++, C#, and Java: f(x);.

2. Here is the same function in Lisp: (f x).

3. Now you know Lisp. Here ends the tutorial.

That's not much of an exaggeration. The syntax of Lisp is really that simple.

The syntax of Clojure is just a bit more complicated. So let's take the above program apart, one statement at a time.

First there's `defn`, which looks like it is being called as a function. Let's go with that for now. The truth is mostly compatible with that view. So the `defn` "function" defines a new function from its arguments. The functions being defined are named `fibs-work` and `fibs`. The square brackets after the function name enclose the

names of the arguments of the function.[1] So the `fib` function takes a single argument named `n`, while the `fibs-work` function takes three arguments named `n`, `i`, and `fs`.

---

1. Actually, the square brackets are Clojure syntax for a "vector" (an array). In this case, that vector contains the symbols that represent the arguments.

Following the argument list is the body of the function. So the body of the `fibs` function is a call to the `cond` function. Think of `cond` like a switch statement that returns a value. The `fibs` function returns the value returned by `cond`.

The arguments to `cond` are a set of pairs. The first element in each pair is a predicate, and the second is the value that `cond` will return if that predicate is `true`. The `cond` function walks down the list of pairs until it sees a true predicate, and then it returns the corresponding value.

The predicates are just function calls. The `(< n 1)` predicate simply calls the `<` function with `n` and `1`. It returns `true` if `n` is less than one. The `(= n 1)` predicate calls the `=` function, which returns `true` if its arguments are equal. The `:else` predicate is considered `true`.

The value returned by `cond` for the `(< n 1)` predicate is `[]`, an empty vector. If `(= n 1)`, then `cond` returns a vector containing `1`. Otherwise, `cond` returns the value produced by the `fibs-work` function.

So, the `fibs` function returns `[]` if `n` is less than `1`, `[1]` if `n` is equal to `1`, and `(fibs-work n 2 [1 1])` in every other case.

Got it? Make sure you do. Go back over it until you do.

The `)))` at the end of the `fibs` function are just the closing parentheses of the `defn`, `cond`, and `fibs-work` function calls. I could have written `fibs` like this:

```
(defn fibs [n]
  (cond
    (< n 1) []
    (= n 1) [1]
    :else (fibs-work n 2 [1 1])
  )
)
```

Perhaps that makes you feel better. Perhaps that relieves the eye-strain headache you felt coming on. And indeed, many new Lisp programmers use this technique to reduce their parentheses anxiety.

That's certainly what I did a decade and a half ago when I first started learning Clojure.

After a few years, however, it becomes obvious that there is no reason to put trailing parentheses on their own lines, and the technique simply becomes an annoyance. Trust me. You'll see.

Anyway, that brings us to the heart of the matter, the `fibs-work` function. If you have gotten comfortable with the `fibs` function, you have probably already worked out most of the details of the `fibs-work` function. But let's go through it step by step just to be sure.

First, the arguments: `[n i fs]`. The `n` argument tells us how many Fibonacci numbers to return. The `i` argument is the index of the next Fibonacci number to compute. The `fs` argument is the current list of Fibonacci numbers.

The `if` function is a lot like the `cond` function. Think of `(if p a b)` as `(cond p a :else b)`. The `if` function takes three arguments. It evaluates the first as a predicate. If the predicate is true, it returns the second argument; otherwise, it returns the third.

So, if `(= i n)`, then we return `fs`. Otherwise… Well, let's walk through that one carefully.

```
  (fibs-work n (inc i) (conj fs (apply + (take-last
```

This is a recursive call to `fibs-work`, passing in `n` unchanged, `i` incremented by one, and `fs` with a new Fibonacci number appended.

It is the `conj` function that does the appending. It takes two arguments: a vector and the value to append to that vector. Vectors are a kind of list. We'll talk about them later.

The `take-last` function takes two arguments: a number `n` and a list. It returns a list containing the last `n` elements of the list argument.

The `apply` function takes two arguments: a function and a list. It calls the function with the list as its arguments. So, `(apply + [3 4])` is equivalent to `(+ 3 4)`.

OK, so now you should have a good working grasp of Clojure. There's more to the language that we'll encounter as we go along. But for now, let's get back to the topic of iteration and recursion.

Iteration

Notice that the recursive call to `fibs-work` is a tail call. The very last thing done by the `fibs-work` function is to call itself. Therefore, the language can employ TCO to eliminate previous stack frames

and turn the recursive call into a `goto`, effectively converting the recursion to pure iteration.

So, then, functions that employ tail calls are, for all intents and purposes, iterative.

TCO, Clojure, and the JVM

The JVM does not make it easy for languages to employ TCO. Indeed, the code I just showed you does not use TCO and therefore grows the stack throughout the iteration. Thus, in Clojure, we *explicitly* invoke TCO by using the `recur` function as follows:

```
(defn fibs-work [n i fs]
  (if (= i n)
    fs
    (recur n (inc i) (conj fs (apply + (take-last
```

The `recur` function can only be called from a tail position, and it effectively reinvokes the enclosing function without growing the stack.

Recursion

There is a much more natural and elegant way to write the Fibonacci algorithm using true recursion:

```
(defn fib [n]
  (cond
    (< n 1) nil
    (<= n 2) 1
    :else (+ (fib (dec n)) (fib (- n 2)))))
(defn fibs [n]
  (map fib (range 1 (inc n))))
```

The `fib` function should be self-explanatory by now. After all, *fib(n)* is just *fib(n–1) + fib(n–2)*. Notice, however, that the calls to `fib` are not on the tail of the function. The last thing executed by the `:else` clause is the `+` function. This means that we cannot use the `recur` function and that TCO is not possible. This also means that the stack will grow as the algorithm proceeds.

The `range` function takes two arguments, *a* and *b*, and returns a list of all the integers from *a* to *b–1*. The `map` function takes two arguments, *f* and *l*. The *f* argument must be a function and the *l* argument must be a list. It calls *f* with each member of *l* and returns a list containing the results.

This version of `fib` is extraordinarily inefficient. Consider this execution profile:

```
fib 20 = 6765
"Elapsed time: 1.459277 msecs"
fib 25 = 75025
"Elapsed time: 11.735279 msecs"
fib 30 = 832040
"Elapsed time: 106.490355 msecs"
fib 34 = 5702887
"Elapsed time: 735.689834 msecs"
```

I didn't bother to analyze the algorithm. But a quick curve fit suggests that the algorithm is $O(n^3)$. So, as elegant as the implementation appears, it will never do.

We can vastly improve the performance by using iteration as follows:

```
(defn ifib
  ([n a b]
   (if (= 0 n)
     b
     (recur (dec n) b (+ a b))))

  ([n]
   (cond
     (< n 1) nil
     (<= n 2) 1
     :else (ifib (- n 2) 1 1)))
  )
```

The `ifib` function has two overloads: `[n a b]` and `[n]`. Since it is iterative, it does not grow the stack, and it is also much faster than the previous recursive version. Indeed, I believe most of that time was spent in printing, rather than true computation.

```
ifib 20 = 6765
"Elapsed time: 0.185508 msecs"
ifib 25 = 75025
"Elapsed time: 0.177111 msecs"
ifib 30 = 832040
"Elapsed time: 0.14596 msecs"
ifib 34 = 5702887
"Elapsed time: 0.148221 msecs"
```

Of course, we've lost a lot of the expressive power of the recursive algorithm. We can reclaim that by remembering *referential transparency*: In a functional language, functions always return the same values given the same inputs. Thus, it is never necessary to reevaluate a function. Once we have computed the value of `(fib 20)`, we can remember it instead of recomputing it.

We do this by using the `memoize` function as follows:

```
(declare fib)
(defn fib-w [n]
```

```
    (cond
      (< n 1) nil
      (<= n 2) 1
      :else (+ (fib (dec n)) (fib (- n 2)))))
  (def fib (memoize fib-w))
```

The `declare` function creates an unbound symbol, which can be used by other functions so long as it is bound before its use. I used `declare` in this case because the definition of `fib` comes after `fib-w`, and Clojure wants all names declared or defined before they are used.

The `memoize` function takes an argument *f*, which must be a function, and returns a new function *g*. Calls to *g* with argument *x* will call *f* with *x* if, and only if, *g* has never been called with *x* before. It then remembers those arguments and the return value. Any subsequent call to *g* with *x* will return the remembered value.

This version of the algorithm is just as fast as the iterative version because we have short-circuited the vast majority of the recursion without sacrificing the elegance of the algorithm. We pay for that with a little extra memory, but that seems a small price to pay.

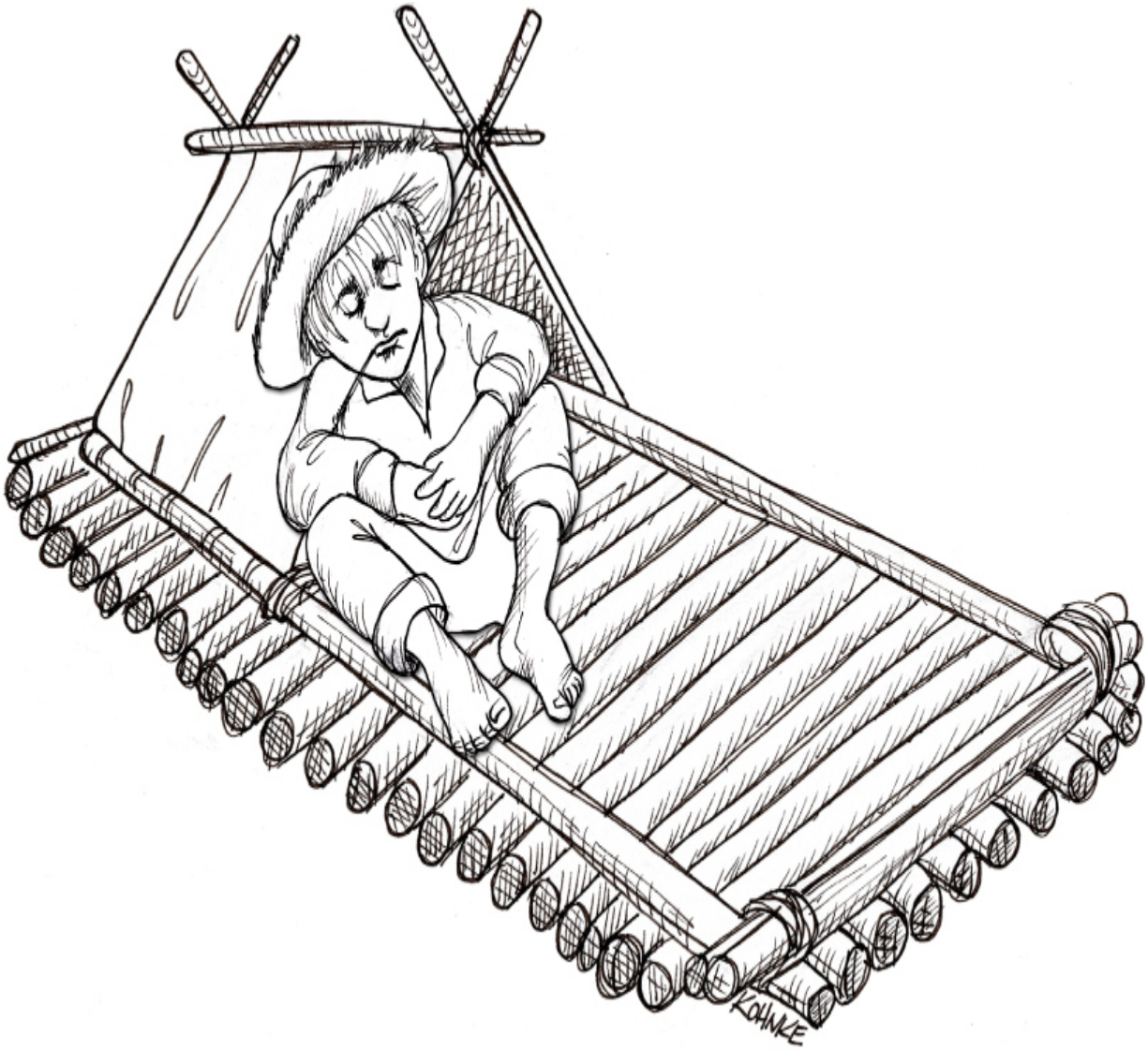
```
fib 20 = 6765
"Elapsed time: 0.168678 msecs"
fib 25 = 75025
```

```
"Elapsed time: 0.16232 msecs"
fib 30 = 832040
"Elapsed time: 0.151619 msecs"
fib 34 = 5702887
"Elapsed time: 0.15134 msecs"
```

What we have learned here is that iteration and recursion are very different approaches. Iterative functions must use tail calls to drive the iteration and should use TCO to prevent the growth of the stack. Recursive functions do not use tail calls and therefore will grow the stack. Truly recursive functions can be quite elegant, and memoization can be used to prevent that elegance from significantly affecting performance.

Although Clojure was used as the language in this chapter, the concepts are the same in virtually every other functional language, and could even be implemented in nonfunctional languages, though with a substantial loss of elegance. ;-)

4. Laziness



Consider the following boldfaced change to our program that calculates a list of Fibonacci numbers:

```
(declare fib)
(defn fib-w [n]
  (cond
```

```
    (< n 1) nil
    (<= n 2) 1
    :else (+ (fib (dec n)) (fib (- n 2)))))
(def fib (memoize fib-w))
(defn lazy-fibs []
  (map fib (rest (range)))

  )
```

The `lazy-fibs` function may look a little strange to you. Let's walk through it. You already understand the `map` function. The `rest` function takes a list and returns that list without the first element. And that brings us to the `range` function.

The `range` function, as called here, returns a list of integers starting at zero. How many integers, you ask? As many as you need. The `range` function is *lazy*. Or, rather, the range function returns a *lazy* list.

What is a lazy list? A lazy list is an object that knows how to compute its next value. In Java, C++, and C#, we called such objects *iterators*. A lazy list is an iterator masquerading as a list.

Clojure is friends with lazy lists. Most of the library functions return lazy lists if possible. So, in the above program, `rest` and `map` both

return a lazy list. And that means that `lazy-fibs` also returns a lazy list.

How would you use `lazy-fibs`? Like so:

```
(take 10 (lazy-fibs))
returns: (1 1 2 3 5 8 13 21 34 55)
```

The `take` function takes two arguments: a number `n` and a list. It returns a list that contains the first `n` elements of the argument list. Actually, that's not quite right, but I'll get to that in a minute.

So, now let's walk through `lazy-fibs` again. The `range` function returns a lazy list of integers starting at zero. The `rest` function takes that list, drops the first element, and then returns a lazy list of the remaining integers, which, in this instance, are the integers starting at one. The `map` function applies each of those integers to the `fib` function returning a lazy list of the Fibonacci numbers starting at `(fib 1)`.

You can have as many Fibonacci numbers as you like, so long as there are no overflows or other machine limitations. So, for example:

```
(nth (lazy-fibs) 50)
returns: 20365011074
```

The `nth` function takes a list and an integer `n` and returns the `nth` element of the list. So this returns the 50th Fibonacci number.

Now consider this:

```
(def list-of-fibs (lazy-fibs))
```

The `def` function (it's not really a function, but pretend that it is) creates a new symbol and associates it with a value. So the symbol `list-of-fibs` refers to a lazy list of Fibonacci numbers, as you can see from the following:

```
(take 5 list-of-fibs)
returns: (1 1 2 3 5)
```

Now note: When we executed the `def` that created `list-of-fibs`, no Fibonacci numbers were calculated, and no memory was allocated for Fibonacci numbers. The calculations only take place, and the memory is only allocated, as the elements of the list are accessed. Remember, behind the scenes, the lazy lists are really just iterators that know how to calculate their next element. Once that calculation takes place, the memory is allocated and the value is placed into a real list.[1]

1. That's a convenient way to think of it for now. Actually, as we'll see shortly, the memory is only allocated, and the list only grows, if the program needs to hold those values.

It is tempting to think of lazy lists as being infinite. Of course, they are not. They are simply unbounded. You can walk through as many items as you like, but that number will always be finite.

Lazy Accumulation

It should be clear that if you continue to pass lazy lists through functions like `map`, `rest`, and `take` (yes, `take` actually returns a lazy list), you will accumulate a long chain of iterators behind the scenes. Each of those iterators must hold on to the function that calculates its next value. It must also hold on to all the data required for that calculation.

I have written applications that have lists with thousands of elements, each of which holds on to other lists with thousands of other elements; and all these lists are lazy. Now remember, we are *deferring* calculations. None of the calculations take place until the final results are accessed. So a huge backlog of deferred iterators gets chained through all those lists.

This works fine until you run out of the memory allocated for holding all those deferred iterators. So, from time to time, it might be a good idea to convert your lazy lists into real lists. In Clojure, we do that with the `doall` function:

```
(def real-list-of-fibs (doall (take 50 (lazy-fibs
```

The `doall` function makes `real-list-of-fibs` a real list that occupies memory and contains no deferred iterators. All calculations have been done.

OK, but Why?

Good question! Laziness is not free. It requires memory and cycles to defer calculations. Then there's the problem of accumulation that can lead to memory exhaustion.

Yet, despite these costs, laziness is a common, if not universal, feature in functional languages. Some languages, like Haskell, are intrinsically lazy. Clojure is not intrinsically lazy, but so many of the library functions are lazy that you cannot easily avoid the laziness. F# and Scala allow laziness, but you must be explicit about it.

Why? Why do all these languages accept the costs of laziness?

Because laziness decouples *what* you need to do from *how much* you need to do. You can write a program that creates a lazy sequence without knowing how big a sequence your users are going to want. Your users can determine how much of your sequence they need.

So, for example:

```
(nth (lazy-fibs) 500)
```

returns

```
225591516161936330872512695036072072046011324913
```

Since `lazy-fibs` puts no limit on the number of Fibonacci numbers it creates, you can ask for as many as you like.

Or, consider this example. I could create a list of $0..n$ integers like this:

```
(range 51)
```

Or like this:

```
(take 51 (range))
```

Notice that in the first example, the 51 is far more coupled than in the second. In the first, I have to get that 51 into the `range` function somehow. I might be able to pass it as an argument, but that's a pretty strong coupling. In the second example, the `range` function doesn't care at all. That 51 could be way out in some other part of the code, far removed from the call to `range`.

By the way, you might be interested to know that in the `lazy-fibs` example above, `(fib 1)` through `(fib 499)` have likely been garbage-collected. Since I'm not holding on to the list itself, the runtime system is free to dispose of the previously calculated elements. Thus, it would be possible to create and traverse a lazy list with trillions of elements and yet never hold more than one[2] of them in memory at a time.

---

2. Or at least some `n`, where `n` is small and is the "chunk" size of the lazy engine.

Coda

There is much more to learn about laziness. My purpose here has been to make you aware of it because it is so common in functional languages. We will be seeing much more of it in the pages to come, but it will almost always be in the background.

5. Statefulness

In the end, every program ever written is just a form of $y = f(x)$, where x is all the input you give to the program and y is all the output it delivers in response.

This definition is sufficient for all batch jobs. For example, in a payroll system, the input *x* is all the employee records and time cards and the output *y* is all the paychecks and reports.

But perhaps this batch definition is too simplistic. After all, in interactive applications, the input you give to the program is often based on the output it just gave you. So perhaps we should think of interactive software systems as:

```
void p(Input x) {
  while (x != DONE)
    x = (getInput(f(x))
}
```

In other words, our program is a loop that computes *y = f(x)* and then hands *y* to some source of input that is passed back into *f* until *f* finally returns `DONE`.

In some very real sense, the state of this program, during each iteration, is *x*. If you were debugging some malfunction, you would want to know the value of *x* and would likely call *x* the state of the system.

And indeed, in the program above, there is a variable named `x` that holds the state of the system and is updated upon each iteration.

However, we can eliminate that variable by writing the program "functionally" as follows:

```
void p(Input x) {
  if (x!=DONE)
    p(getInput(f(x)));
}
```

Now this program has no variable that is updated to hold the state of the system. Instead, that state is passed as an argument from one invocation of `p` to the next.

A few years ago I wrote a functional program in Clojure that looked very much like this. It was a version of the old computer game *Spacewar!*. You can see (and play) this program at https://github.com/unclebob/spacewar. The game is visual and interactive, and it is written in the "functional" style.

The internal state of the `spacewar` program is enormously complex. It consists of the *Enterprise*, dozens of Klingons, hundreds of stars, many dozens of torpedoes, phaser blasts, kinetic projectiles, bases, transports, and a plethora of other entities and attributes. All that complexity is maintained within a single object that I called `world`. And the flow of `spacewar` is, for all intents and purposes:

```
(defn spacewar [world]
  (when (:done? world)
    (System/exit 0))
  (recur (update-world world (get-input world))))
```

In other words, the `spacewar` program is a loop that exits if the `:done?`[1] attribute of the `world` is `true`, and otherwise presents the `world` to the user and gets input that it uses to update the `world`.

---

1. Keywords in Clojure are prefixed with colons. So `:done?` is a keyword, which is just a constant that can be used as an identifier. Often, they are used as keys into hash maps. When used as a function, a keyword behaves like an accessor into a hash map. Thus, `(:done? world)` simply returns the `:done?` element of the `world` hash map.

Here is the actual `update-world` function as it currently exists within `spacewar`:

```
(defn update-world [ms world]
  ;{:pre [(valid-world? world)]
  ; :post [(valid-world? %)]}
```

```
    (->> world
         (game-won ms)
         (game-over ms)
         (ship/update-ship ms)
         (shots/update-shots ms)
         (explosions/update-explosions ms)
         (clouds/update-clouds ms)
         (klingons/update-klingons ms)
         (bases/update-bases ms)
         (romulans/update-romulans ms)
         (view-frame/update-messages ms)
         (add-messages)
         ))
```

The threading macro (`->>`) simply passes the argument `world` into `game-won`, the output of which gets passed to `game-over`, the output of which gets passed to `ship/update-ship`, and so on. Each of those functions returns an updated version of the `world`.

Note the `ms` argument. It contains the number of milliseconds since the last update and is the primary input to the game as a whole. As an object moves across the screen, its position is updated based upon its velocity vector and the number of milliseconds that have transpired since its position was last updated.

I'm showing this to you to give you a glimpse of the complexity being managed by this program. Keep in mind that the `world` is not a mu-

table variable. Each of those threaded functions into which the `world` is being passed is returning a new version of the `world` and passing it to the next. It is not being held in a variable and being mutated.

Let me give you one more glimpse of the complexity:

```
(s/def ::ship (s/keys :req-un
                      [::x ::y ::warp ::warp-char
                       ::impulse ::heading ::velo
                       ::selected-view ::selected
                       ::selected-engine ::target
                       ::engine-power-setting
                       ::weapon-number-setting
                       ::weapon-spread-setting
                       ::heading-setting
                       ::antimatter ::core-temp
                       ::dilithium ::shields
                       ::kinetics ::torpedos
                       ::life-support-damage ::hu

                       ::sensor-damage ::impulse-
                       ::warp-damage ::weapons-da
                       ::strat-scale
                       ::destroyed
                       ::corbomite-device-instal
```

What you are looking at is a small portion of the type specification of the *Enterprise*, the player's ship. Clojure provides a mechanism called `clojure.spec` that give us the ability to very specifically design our data structures with even more precision and control than most statically typed languages.

All this complexity of state is managed within the `spacewar` program by passing the `world` from function to function to function, and then recursively passing it back to `spacewar`. The `world` is never held in a variable.

And, the game operates on a large screen at 30 frames per second.

The bottom line here is that there is no level of complexity that demands that we abandon immutability and deviate from the functional style. On the other hand, there are other factors that do, from time to time, make that demand.

When We MUST Mutate

The `spacewar` program uses a GUI framework called *Quil*.[2] This framework allows the programs that use it to be written in a "functional" style. It may not actually be functional in its internals, but from the outside looking in, there need not be any visible mutable state.

2. See www.quil.info. Quil uses *Processing* behind the scenes. *Processing* is a Java framework that is certainly not functional. Quil pretends to be functional by hiding the mutable variables, or at least by not forcing you to mutate those variables.

On the other hand, I am currently writing an application in Clojure named `more-speech`[3] that uses Java's Swing framework. Swing *is not functional*. Mutable state drips from every appendage of the framework. It is a definitionally mutable object framework.

3. https://github.com/unclebob/more-speech

This makes it a challenge to use with Clojure and maintain a "functional" style. To make matters worse, Swing uses a model-view approach, and the models are defined and controlled by Swing. So building an immutable model is virtually impossible.

Swing is not the only framework that forces you into the mutable world. There are many others. So, even if you are determined to use the "functional" style, you must be able to deal with the fact that a large panoply of existing software frameworks will force you out of that style.

Worse, many such frameworks also force you into the multithreaded world. Swing, for example, runs in its own special thread. Programmers should not use that thread for regular processing but must specifically enter that thread when mutating Swing data structures.

This puts the users of such frameworks into the double jeopardy of mutating state from within multiple threads. The dreaded result of that, of course, is race conditions and concurrent update anomalies.

Fortunately, there are functional languages that provide facilities that reduce the problems of mutation and allow the functional style to interface tolerably well with the multithreaded, nonfunctional style.

Software Transactional Memory (STM)

STM is a set of mechanisms that treat internal memory as though it were a transactional commit/rollback database. The transactions are functions that are protected from concurrent update by a *compare-and-swap* protocol.

If that was too much of a word salad, perhaps an example would be clarifying.

Let us say that we have an object *o* and a function *f* that mutates *o*. So *of = f(o)* where *of* is the original *o* mutated by *f*.

The problem is that *f* takes time to do its work, and there is a chance that some other thread will interrupt *f* and apply its own operation *g* on *o*: *og = g(o)*. When *f* finally completes, what is the state of *o*? Is it *of*? Or is it *og*? Or have both mutations been applied, giving us *of g*?

The typical concurrent update problem would most often yield *of*, causing the operation of *g* to be lost. Programmers often resolve this kind of problem by *locking o* so that *g* cannot interrupt *f*, and vice versa. The lock forces the interrupting thread to wait until *o* is unlocked. The problem, however, is that this can lead to the dreaded *deadly embrace*.[4]

---

4. Sometimes known as *deadlock*.

Imagine that we have two objects *o* and *p* and two functions *f(o, p)* and *g(p, o)*. These functions lock their arguments before operating on them. Suppose *f* and *g* are executing in different threads and *g* interrupts *f* just after *f* locks *o*. Now *g* locks *p* but cannot lock *o* because *o* is locked by *f*, so *g* waits. Now *f* wakes up and tries to lock *p* but cannot because *p* is locked by *g*—and nothing can proceed. The functions *f* and *g* are in a deadly embrace.

The problem of deadly embrace can be avoided by locking everything in the same order every time. If *f* and *g* agree to lock *o* first and *p* second, then the embrace cannot happen. However, these agreements

are hard to enforce, and as systems get more and more complicated, a correct locking order can be very difficult to divine.

STM solves this problem by *not* locking, and instead using a commit/rollback technique. Let's call this technique *swap*. We can enact it with *swap(o, f)*, which will hold the current value of *o* in *oh*, compute *of = f(o)*, and then, in an *atomic*[5] operation, compare the current value of *o* with *oh* and, if they are the same, swap *o* with *of*. If the compare fails, then the operation is repeated from the beginning and will continue repeating until the compare succeeds.

---

5. Atomic operations cannot be interrupted.

There are several ways to use STM in Clojure, but the simplest is the `atom`. An `atom` is an *atomic* value that can be altered using the `swap!` function. Here's an example:

```
(def counter (atom 0))
(defn add-one [x]
  (let [y (inc x)]

    (print (str "(" x ")"))
    y))
(defn increment [n id]
  (dotimes [_ n]
```

```
    (print id)
    (swap! counter add-one)))
(defn -main []
  (let [ta (future (increment 10 "a"))

        tx (future (increment 10 "x"))
        _ @ta
        _ @tx]
    (println "\nCounter is: " @counter)))
```

The first line creates the `atom` named `counter`. The `-main` program starts two threads, using `future`, both of which call the `increment` function. The `@ta` and `@tx` expressions wait for the respective threads to complete.

The `add-one` function adds one to its argument, but that `print` function can allow another thread to jump in; and that's exactly what happens. Here's an example of the output:

```
a(0)a(1)a(2)a(3)a(4)xa(5)x(5)(6)(6)x(7)(7)a(8)(8)
x(9)(9)a(10)(10)x(11)a(11)(12)(12)a(13)x(13)(14)(
x(15)(15)(16)x(17)x(18)x(19) Counter is: 20
```

At first, thread `a` runs without interruption for a while. But at the fifth increment, the `x` thread jumps in, and the two fight each other. No-

tice the repeated values as the `swap!` detects the collisions and repeats. Finally, thread `a` finishes and thread `x` experiences no further interruptions. The end count of 20 is correct.

Life Is Hard, Software Is Harder

It would be nice to live, full time, in a functional world. Multiple threads in a functional world generally do not have race conditions.[6] After all, if you never update, you can't have concurrent update problems. But all too often we are forced back into the multithreaded, nonfunctional world by frameworks, or legacy code. And when that happens, the mechanisms of STM can help us avoid the worst of an otherwise horrific situation.

---

6. See Chapter 15, Concurrency, for when they do.

II: Comparative Analysis

6. Prime Factors [This content is currently in development.]

This content is currently in development.

7. Bowling Game [This content is currently in development.]

This content is currently in development.

8. Gossiping Bus Drivers [This content is currently in development.]

This content is currently in development.

9. Object Oriented Programming [This content is currently in development.]

This content is currently in development.

10. Types [This content is currently in development.]

This content is currently in development.

III: Functional Design

11. Data Flow [This content is currently in development.]

This content is currently in development.

12. SOLID [This content is currently in development.]

This content is currently in development.

IV: Functional Pragmatics

13. Tests [This content is currently in development.]

This content is currently in development.

14. GUI [This content is currently in development.]

This content is currently in development.

15. Concurrency [This content is currently in development.]

This content is currently in development.

V: Design Patterns

16. Design Patterns [This content is currently in development.]

This content is currently in development.

VI: Case Study

17. Case Study: Wator [This content is currently in development.]

This content is currently in development.

Afterword [This content is currently in development.]

This content is currently in development.