Mastering Clojure

Clojure is a general-purpose language from the Lisp family with an emphasis on functional programming. It has some interesting concepts, such as immutability, gradual typing, thread-safe concurrency primitives, and macro-based metaprogramming, which help to create modern, performant, and scalable applications.

Mastering Clojure will start off by exploring the details of sequences, concurrency primitives, and macros, followed by parallel, asynchronous, and reactive programming techniques. Packed with examples, this book will give you a walkthrough of orchestrating concurrency and parallelism. Later on, we'll discuss the advantages of Clojure's powerful macro system and how it also supports other programming paradigms, such as pure functional programming and logic programming. We also explain how reducers and transducers can be used to handle data in a more performant manner. Lastly, we'll show you how to test and troubleshoot your code to allow you to deploy the code faster.

Who this book is written for

If you're looking to learn more about the core libraries and dive deep into the Clojure language, then this book is ideal for you. Prior knowledge of the Clojure language is required.

What you will learn from this book

- Maximize the impact of parallelization, functional composition, and process transformation by composing reducers and transducers
- Process and manipulate data using sequences, reducers, and transducers in Clojure
- Modify and add features to the Clojure language using macros
- Explore concepts from category theory and logic programming in Clojure
- Orchestrate parallelism and concurrency using built-in primitives as well as community libraries in Clojure
- Handle data with asynchronous and reactive programming methodologies and leverage it using the core.async library
- Test your code with unit tests, specs, and type checks
- Troubleshoot and style your code to make it more maintainable

Understand the philosophy of the Clojure language and dive into its inner workings to unlock its advanced features, methodologies, and constructs.
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Working with Sequences and Patterns'
- A synopsis of the book’s content
- More information on Mastering Clojure
Akhil Wali is a software developer. He has been writing code as a hobbyist since 1997 and professionally since 2010. He completed his post graduation from Santa Clara University in 2010, and he graduated from Visvesvaraya Technological University in 2008. His areas of work include business intelligence systems, ERP systems, search engines, and document collaboration tools. He mostly works with Clojure, JavaScript, and C#. Apart from computers, his interests include soccer, guitar solos, and finding out more about the universe.
Ever since the dawn of computers decades ago, there have been a number of programming languages created for the purpose of writing software. One of the earliest of these languages is Lisp, whose name is an abbreviation of the term "list processing". Lisp has evolved greatly over time, and there are now several dialects of Lisp. Each of these dialects emphasizes its own set of ideas and features. Clojure is one among these Lisps, and it focuses on immutability, concurrency, and parallelism. It emphasizes being simple, practical, and intuitive, which makes it easy to learn. It is said that you have never realized how a language can be powerful until you have programmed in a Lisp, and Clojure is no exception to this rule. A skilled Clojure programmer can easily and quickly create software that is both performant and scalable.

With the recent rise of parallel data processing and multicore architectures, functional programming languages have become more popular for creating software that is both provable and performant. Clojure brings functional programming to the Java Virtual Machine (JVM), and also to web browsers through ClojureScript. Like other functional programming languages, Clojure focuses on the use of functions and immutable data structures for writing programs. Clojure also adds a hint of Lisp through the use of symbolic expressions and a dynamic type system.

This book will walk you through the interesting features of the Clojure language. We will also discuss some of the more advanced and lesser known programming constructs in Clojure. Several libraries from the Clojure ecosystem that we can put to practical use in our own programs will also be described. You won't need to be convinced any more about the elegance and power of the Clojure language by the time you've finished this book.

This book wouldn't have materialized without the feedback from the technical reviewers and the effort of the content and editing teams at Packt Publishing.
What this book covers

Chapter 1, Working with Sequences and Patterns, describes several elementary programming techniques, such as recursion, sequences, and pattern matching.

Chapter 2, Orchestrating Concurrency and Parallelism, explains the various constructs available in the Clojure language for concurrent and parallel programming.

Chapter 3, Parallelization Using Reducers, introduces reducers, which are abstractions of collection types for parallel data processing.

Chapter 4, Metaprogramming with Macros, explains how we can use macros and quoting to implement our own programming constructs in Clojure.

Chapter 5, Composing Transducers, describes how we can define and compose data transformations using transducers.

Chapter 6, Exploring Category Theory, explores algebraic data structures, such as functors, monoids, and monads, from the pure functional programming world.

Chapter 7, Programming with Logic, describes how we can use logical relations to solve problems.

Chapter 8, Leveraging Asynchronous Tasks, explains how we can write code that is executed asynchronously.

Chapter 9, Reactive Programming, describes how we can implement solutions to problems using asynchronous event streams.

Chapter 10, Testing Your Code, covers several testing libraries that are useful in verifying our code. This chapter describes techniques such as test-driven development, behavior-driven development, and generative testing.

Chapter 11, Troubleshooting and Best Practices, describes techniques to debug your code as well as several good practices for developing Clojure applications and libraries.
In this chapter, we'll revisit a few basic programming techniques, such as recursion and sequences, with Clojure. As we will see, Clojure focuses on the use of higher-order functions to abstract computation, like any other functional programming language. This design can be observed in most, if not all, of the Clojure standard library. In this chapter, we will cover the following topics:

- Exploring recursion
- Learning about sequences and laziness
- Examining zippers
- Briefly studying pattern matching

Defining recursive functions

Recursion is one of the central methodologies of computer science. It allows us to elegantly solve problems that have cumbersome non-recursive solutions. Yet, recursive functions are discouraged in quite a few imperative programming languages in favor of non-recursive functions. Clojure does no such thing and completely embraces recursion along with all its pros and cons. In this section, we will explore how to define recursive functions.

The following examples can be found in src/m_clj/c1/recur.clj of the book’s source code.
In general, a function can be made recursive by simply calling it again from within the body of the function. We can define a simple function to return the first $n$ numbers of the Fibonacci sequence as shown in Example 1.1:

```
(defn fibo
  ([n]
   (fibo [0N 1N] n))
  ([xs n]
   (if (<= n (count xs))
     xs
     (let [x' (+ (last xs)
                 (nth xs (- (count xs) 2)))]
       (conj xs x')
       (fibo xs' n)))))
```

Example 1.1: A simple recursive function

The Fibonacci sequence is a series of numbers that can be defined as follows:

The first element $F_0$ is 0 and the second element $F_1$ is 1.

The rest of the numbers are the sum of the previous two numbers, that is the $n$th Fibonacci number $F_n = F_{n-1} + F_{n-2}$.

In the previously defined `fibo` function, the last two elements of the list are determined using the `nth` and `last` functions, and the sum of these two elements is appended to the list using the `conj` function. This is done in a recursive manner, and the function terminates when the length of the list, determined by the `count` function becomes equal to the supplied value $n$. Also, the values `0N` and `1N`, which represent `BigInteger` types, are used instead of the values 0 and 1. This is done because using long or integer values for such a computation could result in an arithmetic overflow error. We can try out this function in the REPL shown as follows:

```
user> (fibo 10)
[0N 1N 1N 2N 3N 5N 8N 13N 21N 34N]
user> (last (fibo 100))
218922995834555169026N
```

The `fibo` function returns a vector of the first $n$ Fibonacci numbers as expected. However, for larger values of $n$, this function will cause a stack overflow:

```
user> (last (fibo 10000))
StackOverflowError  clojure.lang.Numbers.lt (Numbers.java:219)
```
The reason for this error is that there were too many nested function calls. A call to any function requires an additional call stack. With recursion, we reach a point where all of the available stack space in a program is consumed and no more function calls can be performed. A tail call can overcome this limitation by using the existing call stack for a recursive call, which removes the need for allocating a new call stack. This is only possible when the return value of a function is the return value of a recursive call made by the function, in which case an additional call stack is not required to store the state of the function that performs the recursive call. This technique is termed as tail call elimination. In effect, a tail call optimized function consumes a constant amount of stack space.

In fact, the fibo function does indeed make a tail call, as the last expression in the body of the function is a recursive call. Still, it consumes stack space for each recursive call. This is due to the fact that the underlying virtual machine, the JVM, does not perform tail call elimination. In Clojure, tail call elimination has to be done explicitly using a recur form to perform a recursive call. The fibo function we defined earlier can be refined to be tail recursive by using a recur form, as shown in Example 1.2:

```
(defn fibo-recur
    ([n]
        (fibo-recur [0N 1N] n))
    ([xs n]
        (if (<= n (count xs))
            xs
            (let [x' (+ (last xs) (nth xs (- (count xs) 2)))
                  xs' (conj xs x')]
                (recur xs' n)))))
```

Example 1.2: A recursive function defined using recur

**Downloading the example code**

You can download the example code files from your account at http://www.packtpub.com for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed to you.
Effectively, the `fibo-recur` function can perform an infinite number of nested recursive calls. We can observe that this function does not blow up the stack for large values of `n`, shown as follows:

```
user> (fibo-recur 10)
[0N 1N 1N 2N 3N 5N 8N 13N 21N 34N]
user> (last (fibo-recur 10000))
207936...230626N
```

We should note that a call to `fibo-recur` can take quite a while to terminate for large values of `n`. We can measure the time taken for a call to `fibo-recur` to complete and return a value, using the `time` macro, as follows:

```
user> (time (last (fibo-recur 10000)))
"Elapsed time: 1320.050942 msecs"
207936...230626N
```

The `fibo-recur` function can also be expressed using the `loop` and `recur` forms. This eliminates the need for using a second function arity to pass the `[0N 1N]` value around, as shown in the `fibo-loop` function defined in `Example 1.3`:

```
(defn fibo-loop [n]
  (loop [xs [0N 1N] n n]
    (if (<= n (count xs))
      xs
      (let [x' (+ (last xs)
                  (nth xs (- (count xs) 2)))
            xs' (conj xs x')]
        (recur xs' n)))))
```

Note that the `loop` macro requires a vector of bindings (pairs of names and values) to be passed as its first argument. The second argument to the `loop` form must be an expression that uses the `recur` form. This nested `recur` form calls the surrounding expression recursively by passing in the new values for the declared bindings in the `loop` form. The `fibo-loop` function returns a value that is equal to that returned by the `fibo-recur` function, from `Example 1.2`, shown as follows:

```
user> (fibo-loop 10)
[0N 1N 1N 2N 3N 5N 8N 13N 21N 34N]
user> (last (fibo-loop 10000))
207936...230626N
```
Another way to handle recursion is by using the trampoline function. The trampoline function takes a function as its first argument, followed by the values of the parameters to be passed to the supplied function. A trampoline form expects the supplied function to return another function, and in such a case, the returned function will be invoked. Thus, a trampoline form manages recursion by obtaining a return value, and invoking the returned value again if it's a function. Thus, the trampoline function avoids using any stack space. Each time the supplied function is invoked, it returns and the result gets stored in the process heap. For example, consider the function in Example 1.4 that calculates the first \( n \) numbers of the Fibonacci sequence using a trampoline:

```clojure
(defn fibo-trampoline [n]
  (letfn [[(fibo-fn [xs n]]
    (if (<= n (count xs))
      xs
      (let [x' (+ (last xs)
                  (nth xs (- (count xs) 2)))]
        xs' (conj xs x'))
      #(fibo-fn xs' n)))
    (trampoline fibo-fn [0N 1N] n)))
```

Example 1.4: A recursive function defined using trampoline

In the `fibo-trampoline` function, the internal `fibo-fn` function returns either a sequence, denoted by `xs`, or a closure that takes no arguments, represented by `#(fibo-fn xs' n)`. This function is equivalent to the `fibo-recur` function we defined earlier, even in terms of performance, shown as follows:

```
user> (fibo-trampoline 10)
[0N 1N 1N 2N 3N 5N 8N 13N 21N 34N]
user> (time (last (fibo-trampoline 10000)))
"Elapsed time: 1346.629108 msecs"
207936...230626N
```

Mutual recursion can also be handled effectively using a trampoline. In mutual recursion, two functions call each other in a recursive manner. For example, consider the function that utilizes two mutually recursive functions in Example 1.5:

```clojure
(defn sqrt-div2-recur [n]
  (letfn [[sqrt [n]]
    (if (< n 1)
      n
      (div2 (Math/sqrt n)))
    (div2 [n]
      (if (< n 1)
```
n
(sqrt (/ n 2)))
(sqrt n)))

Example 1.5: A simple function that uses mutual recursion

The `sqrt-div2-recur` function from Example 1.5 defines two mutually recursive functions internally, namely `sqrt` and `div2`, that repeatedly square root and halve a given value \( n \) until the calculated value is less than 1. The `sqrt-div2-recur` function declares these two functions using a `letfn` form and invokes the `sqrt` function. We can convert this to use a trampoline form as shown in Example 1.6:

```clojure
defn sqrt-div2-trampoline [n]
  (letfn [(sqrt [n]
            (if (< n 1)
              n
              #(div2 (Math/sqrt n))))
          (div2 [n]
            (if (< n 1)
              n
              #(sqrt (/ n 2)))]
            (trampoline sqrt n)))
```

Example 1.6: A function that uses mutual recursion using trampoline

In the previous `sqrt-div2-trampoline` function shown, the functions `sqrt` and `div2` return closures instead of calling a function directly. The trampoline form in the body of the function calls the `sqrt` function while supplying the value \( n \). Both the `sqrt-div2-recur` and `sqrt-div2-trampoline` functions take about the same time to return a value for the given value of \( n \). Hence, using a trampoline form does not have any additional performance overhead, shown as follows:

```clojure
user> (time (sqrt-div2-recur 10000000000N))
"Elapsed time: 0.327439 msecs"
0.5361105866719398
user> (time (sqrt-div2-trampoline 10000000000N))
"Elapsed time: 0.326081 msecs"
0.5361105866719398
```

As the preceding examples demonstrate, there are various ways to define recursive functions in Clojure. Recursive functions can be optimized using tail call elimination, by using `recur`, and mutual recursion, which is done using the trampoline function.
Thinking in sequences

A **sequence**, shortened as a **seq**, is essentially an abstraction of a list. This abstraction provides a unified model or interface to interact with a collection of items. In Clojure, all the primitive data structures, namely strings, lists, vectors, maps, and sets can be treated as sequences. In practice, almost everything that involves iteration can be translated into a sequence of computations. A collection is termed as **seqable** if it implements the abstraction of a sequence. We will learn everything there is to know about sequences in this section.

Sequences can also be **lazy**. A lazy sequence can be thought of as a possibly infinite series of computed values. The computation of each value is deferred until it is actually needed. We should note that the computation of a recursive function can easily be represented as a lazy sequence. For example, the Fibonacci sequence can be computed by lazily adding the last two elements in the previously computed sequence. This can be implemented as shown in *Example 1.7*.

The following examples can be found in `src/m_clj/c1/seq.clj` of the book’s source code.

```clojure
(defn fibo-lazy [n]
  (->> [0N 1N]
    (iterate (fn [[a b]] [b (+ a b)]))
    (map first)
    (take n)))
```

*Example 1.7: A lazy Fibonacci sequence*

The threading macro `->>` is used to pass the result of a given expression as the last argument to the next expression, in a repetitive manner for all expressions in its body. Similarly, the threading macro `->` is used to pass the result of a given expression as the first argument to the subsequent expressions.

The `fibo-lazy` function from *Example 1.7* uses the `iterate`, `map`, and `take` functions to create a lazy sequence. We will study these functions in more detail later in this section. The `fibo-lazy` function takes a single argument `n`, which indicates the number of items to be returned by the function. In the `fibo-lazy` function, the values `0N` and `1N` are passed as a vector to the `iterate` function, which produces a lazy sequence. The function used for this iteration creates a new pair of values `b` and `(a b)` from the initial values `a` and `b`. 
Next, the `map` function applies the first function to obtain the first element in each resulting vector. A `take` form is finally applied to the sequence returned by the `map` function to retrieve the first \( n \) values in the sequence. The `fibo-lazy` function does not cause any error even when passed relatively large values of \( n \), shown as follows:

```clojure
user> (fibo-lazy 10)
(0N 1N 1N 2N 3N 5N 8N 13N 21N 34N)
user> (last (fibo-lazy 10000))
207936...230626N
```

Interestingly, the `fibo-lazy` function in Example 1.7 performs significantly better than the recursive functions from Example 1.2 and Example 1.3, as shown here:

```clojure
user> (time (last (fibo-lazy 10000)))
"Elapsed time: 18.593018 msecs"
207936...230626N
```

Also, binding the value returned by the `fibo-lazy` function to a variable does not really consume any time. This is because this returned value is lazy and not evaluated yet. Also, the type of the return value is `clojure.lang.LazySeq`, as shown here:

```clojure
user> (time (def fibo-xs (fibo-lazy 10000)))
"Elapsed time: 0.191981 msecs"
#'user/fibo-xs
user> (type fibo-xs)
clojure.lang.LazySeq
```

We can optimize the `fibo-lazy` function even further by using `memoization`, which essentially caches the value returned by a function for a given set of inputs. This can be done using the `memoize` function, as follows:

```clojure
(def fibo-mem (memoize fibo-lazy))
```

The `fibo-mem` function is a memoized version of the `fibo-lazy` function. Hence, subsequent calls to the `fibo-mem` function for the same set of inputs will return values significantly faster, shown as follows:

```clojure
user> (time (last (fibo-mem 10000)))
"Elapsed time: 19.776527 msecs"
207936...230626N
user> (time (last (fibo-mem 10000)))
"Elapsed time: 2.82709 msecs"
207936...230626N
```
Note that the memoize function can be applied to any function, and it is not really related to sequences. The function we pass to memoize must be free of side effects, or else any side effects will be invoked only the first time the memoized function is called with a given set of inputs.

**Using the seq library**

Sequences are a truly ubiquitous abstraction in Clojure. The primary motivation behind using sequences is that any domain with sequence-like data in it can be easily modelled using the standard functions that operate on sequences. This infamous quote from the Lisp world reflects on this design:

"It is better to have 100 functions operate on one data abstraction than 10 functions on 10 data structures."

A sequence can be constructed using the cons function. We must provide an element and another sequence as arguments to the cons function. The first function is used to access the first element in a sequence, and similarly the rest function is used to obtain the other elements in the sequence, shown as follows:

```clojure
user> (def xs (cons 0 '(1 2 3)))
#'user/xs
user> (first xs)
0
user> (rest xs)
(1 2 3)
```

The first and rest functions in Clojure are equivalent to the car and cdr functions, respectively, from traditional Lisps. The cons function carries on its traditional name.

In Clojure, an empty list is represented by the literal (). An empty list is considered as a *truthy* value, and does not equate to nil. This rule is true for any empty collection. An empty list does indeed have a type – it's a list. On the other hand, the nil literal signifies the absence of a value, of any type, and is not a truthy value. The second argument that is passed to cons could be empty, in which case the resulting sequence would contain a single element:

```clojure
user> (cons 0 ())
(0)
user> (cons 0 nil)
(0)
user> (rest (cons 0 nil))
()
```
An interesting quirk is that `nil` can be treated as an empty collection, but the converse is not true. We can use the `empty?` and `nil?` functions to test for an empty collection and a `nil` value, respectively. Note that `(empty? nil)` returns `true`, shown as follows:

```
user> (empty? ())
true
user> (empty? nil)
true
user> (nil? ())
false
user> (nil? nil)
true
```

By the *truthy* value, we mean to say a value that will test positive in a conditional expression such as an `if` or a `when` form.

The `rest` function will return an empty list when supplied an empty list. Thus, the value returned by `rest` is always truthy. The `seq` function can be used to obtain a sequence from a given collection. It will return `nil` for an empty list or collection. Hence, the `head`, `rest` and `seq` functions can be used to iterate over a sequence. The `next` function can also be used for iteration, and the expression `(seq (rest coll))` is equivalent to `(next coll)`, shown as follows:

```
user> (= (rest ()) nil)
false
user> (= (seq ()) nil)
true
user> (= (next ()) nil)
true
```

The `sequence` function can be used to create a list from a sequence. For example, `nil` can be converted into an empty list using the expression `(sequence nil)`. In Clojure, the `seq?` function is used to check whether a value implements the sequence interface, namely `clojure.lang.ISeq`. Only lists implement this interface, and other data structures such as vectors, sets, and maps have to be converted into a sequence by using the `seq` function. Hence, `seq?` will return `true` only for lists. Note that the `list?`, `vector?`, `map?`, and `set?` functions can be used to check the concrete type of a given collection. The behavior of the `seq?` function with lists and vectors can be described as follows:

```
user> (seq? '(1 2 3))
true
user> (seq? [1 2 3])
```
false
user> (seq? (seq [1 2 3]))
true

Only lists and vectors provide a guarantee of sequential ordering among elements. In other words, lists and vectors will store their elements in the same order or sequence as they were created. This is in contrast to maps and sets, which can reorder their elements as needed. We can use the `sequential?` function to check whether a collection provides sequential ordering:

user> (sequential? '(1 2 3))
true
user> (sequential? [1 2 3])
true
user> (sequential? {:a 1 :b 2})
false
user> (sequential? #{:a :b})
false

The `associative?` function can be used to determine whether a collection or sequence associates a key with a particular value. Note that this function returns true only for maps and vectors:

user> (associative? '(1 2 3))
false
user> (associative? [1 2 3])
true
user> (associative? {:a 1 :b 2})
true
user> (associative? #{:a :b})
false

The behavior of the `associative?` function is fairly obvious for a map since a map is essentially a collection of key-value pairs. The fact that a vector is also associative is well justified too, as a vector has an implicit key for a given element, namely the index of the element in the vector. For example, the `[:a :b]` vector has two implicit keys, 0 and 1, for the elements :a and :b respectively. This brings us to an interesting consequence – vectors and maps can be treated as functions that take a single argument, that is a key, and return an associated value, shown as follows:

user> ([:a :b] 1)
:b
user> ([:a :b] 2) :a
1
Although they are not associative by nature, sets are also functions. Sets return a value contained in them, or nil, depending on the argument passed to them, shown as follows:

```
user> (#{1 2 3} 1)
1
user> (#{1 2 3} 0)
nil
```

Now that we have familiarized ourselves with the basics of sequences, let's have a look at the many functions that operate over sequences.

Creating sequences

There are several ways to create sequences other than using the cons function. We have already encountered the conj function in the earlier examples of this chapter. The conj function takes a collection as its first argument, followed by any number of arguments to add to the collection. We must note that conj behaves differently for lists and vectors. When supplied a list, the conj function adds the other arguments at the head, or start, of the list. In case of a vector, the conj function will insert the other arguments at the tail, or end, of the vector:

```
user> (conj [1 2 3] 4 5 6)
[1 2 3 4 5 6]
user> (conj '(1 2 3) 4 5 6)
(6 5 4 1 2 3)
```

The concat function can be used to join or concatenate any number of sequences in the order in which they are supplied, shown as follows:

```
user> (concat [1 2 3] [])
(1 2 3)
user> (concat [] [1 2 3])
(1 2 3)
user> (concat [1 2 3] [4 5 6] [7 8 9])
(1 2 3 4 5 6 7 8 9)
```

A given sequence can be reversed using the reverse function, shown as follows:

```
user> (reverse [1 2 3 4 5 6])
(6 5 4 3 2 1)
user> (reverse (reverse [1 2 3 4 5 6]))
(1 2 3 4 5 6)
```
The range function can be used to generate a sequence of values within a given integer range. The most general form of the range function takes three arguments—the first argument is the start of the range, the second argument is the end of the range, and the third argument is the step of the range. The step of the range defaults to 1, and the start of the range defaults to 0, as shown here:

```
user> (range 5)
(0 1 2 3 4)
user> (range 0 10 3)
(0 3 6 9)
user> (range 15 10 -1)
(15 14 13 12 11)
```

We must note that the range function expects the start of the range to be less than the end of the range. If the start of the range is greater than the end of the range and the step of the range is positive, the range function will return an empty list. For example, (range 15 10) will return (). Also, the range function can be called with no arguments, in which case it returns a lazy and infinite sequence starting at 0.

The take and drop functions can be used to take or drop elements in a sequence. Both functions take two arguments, representing the number of elements to take or drop from a sequence, and the sequence itself, as follows:

```
user> (take 5 (range 10))
(0 1 2 3 4)
user> (drop 5 (range 10))
(5 6 7 8 9)
```

To obtain an item at a particular position in the sequence, we should use the nth function. This function takes a sequence as its first argument, followed by the position of the item to be retrieved from the sequence as the second argument:

```
user> (nth (range 10) 0)
0
user> (nth (range 10) 9)
9
```

To repeat a given value, we can use the repeat function. This function takes two arguments and repeats the second argument the number of times indicated by the first argument:

```
user> (repeat 10 0)
(0 0 0 0 0 0 0 0 0 0)
user> (repeat 5 :x)
(:x :x :x :x :x)
```
The repeat function will evaluate the expression of the second argument and repeat it. To call a function a number of times, we can use the repeatedly function, as follows:

```
user> (repeat 5 (rand-int 100))
(75 75 75 75 75)
user> (repeatedly 5 #(rand-int 100))
(88 80 17 52 32)
```

In this example, the repeat form first evaluates the (rand-int 100) form, before repeating it. Hence, a single value will be repeated several times. Note that the rand-int function simply returns a random integer between 0 and the supplied value. On the other hand, the repeatedly function invokes the supplied function a number of times, thus producing a new value every time the rand-int function is called.

A sequence can be repeated an infinite number of times using the cycle function. As you might have guessed, this function returns a lazy sequence to indicate an infinite series of values. The take function can be used to obtain a limited number of values from the resulting infinite sequence, shown as follows:

```
user> (take 5 (cycle [0]))
(0 0 0 0 0)
user> (take 5 (cycle (range 3)))
(0 1 2 0 1)
```

The interleave function can be used to combine any number of sequences. This function returns a sequence of the first item in each collection, followed by the second item, and so on. This combination of the supplied sequences is repeated until the shortest sequence is exhausted of values. Hence, we can easily combine a finite sequence with an infinite one to produce another finite sequence using the interleave function:

```
user> (interleave [0 1 2] [3 4 5 6] [7 8])
(0 3 7 1 4 8)
user> (interleave [1 2 3] (cycle [0]))
(1 0 2 0 3 0)
```

Another function that performs a similar operation is the interpose function. The interpose function inserts a given element between the adjacent elements of a given sequence:

```
user> (interpose 0 [1 2 3])
(1 0 2 0 3)
```
The iterate function can also be used to create an infinite sequence. Note that we have already used the iterate function to create a lazy sequence in Example 1.7. This function takes a function \( f \) and an initial value \( x \) as its arguments. The value returned by the iterate function will have \((f \ x)\) as the first element, \((f \ (f \ x))\) as the second element, and so on. We can use the iterate function with any other function that takes a single argument, as follows:

```
user> (take 5 (iterate inc 5))
(5 6 7 8 9)
user> (take 5 (iterate #(+ 2 %) 0))
(0 2 4 6 8)
```

**Transforming sequences**

There are also several functions to convert sequences into different representations or values. One of the most versatile of such functions is the map function. This function maps a given function over a given sequence, that is, it applies the function to each element in the sequence. Also, the value returned by map is implicitly lazy. The function to be applied to each element must be the first argument to map, and the sequence on which the function must be applied is the next argument:

```
user> (map inc [0 1 2 3])
(1 2 3 4)
user> (map #(* 2 %) [0 1 2 3])
(0 2 4 6)
```

Note that map can accept any number of collections or sequences as its arguments. In this case, the resulting sequence is obtained by passing the first items of the sequences as arguments to the given function, and then passing the second items of the sequences to the given function, and so on until any of the supplied sequences are exhausted. For example, we can sum the corresponding elements of two sequences using the map and + functions, as shown here:

```
user> (map + [0 1 2 3] [4 5 6])
(4 6 8)
```

The mapv function has the same semantics of map, but returns a vector instead of a sequence, as shown here:

```
user> (mapv inc [0 1 2 3])
[1 2 3 4]
```
Another variant of the `map` function is the `map-indexed` function. This function expects that the supplied function will accept two arguments—one for the index of a given element and another for the actual element in the list:

```clojure
user> (map-indexed (fn [i x] [i x]) "Hello")
([0 \H] [1 \e] [2 \l] [3 \l] [4 \o])
```

In this example, the function supplied to `map-indexed` simply returns its arguments as a vector. An interesting point that we can observe from the preceding example is that a string can be treated as a sequence of characters.

The `mapcat` function is a combination of the `map` and `concat` function. This function maps a given function over a sequence, and applies the `concat` function on the resulting sequence:

```clojure
user> (require '[clojure.string :as cs])
nil
user> (map #(cs/split % #"\d") ["aa1bb" "cc2dd" "ee3ff"])
(["aa" "bb"] ["cc" "dd"] ["ee" "ff"])
user> (mapcat #(cs/split % #"\d") ["aa1bb" "cc2dd" "ee3ff"])
("aa" "bb" "cc" "dd" "ee" "ff")
```

In this example, we use the `split` function from the `clojure.string` namespace to split a string using a regular expression, shown as `#"\d"`. The `split` function will return a vector of strings, and hence the `mapcat` function returns a sequence of strings instead of a sequence of vectors like the `map` function.

The `reduce` function is used to combine or reduce a sequence of items into a single value. The `reduce` function requires a function as its first argument and a sequence as its second argument. The function supplied to `reduce` must accept two arguments. The supplied function is first applied to the first two elements in the given sequence, and then applied to the previous result and the third element in the sequence, and so on until the sequence is exhausted. The `reduce` function also has a second arity, which accepts an initial value, and in this case, the supplied function is applied to the initial value and the first element in the sequence as the first step. The `reduce` function can be considered equivalent to loop-based iteration in imperative programming languages. For example, we can compute the sum of all elements in a sequence using `reduce`, as follows:

```clojure
user> (reduce + [1 2 3 4 5])
15
user> (reduce + [])
0
user> (reduce + 1 [])
1
```
In this example, when the `reduce` function is supplied an empty collection, it returns 0, since (+) evaluates to 0. When an initial value of 1 is supplied to the `reduce` function, it returns 1, since (+ 1) returns 1.

A list comprehension can be created using the `for` macro. Note that a `for` form will be translated into an expression that uses the `map` function. The `for` macro needs to be supplied a vector of bindings to any number of collections, and an expression in the body. This macro binds the supplied symbol to each element in its corresponding collection and evaluates the body for each element. Note that the `for` macro also supports a :let clause to assign a value to a variable, and also a :when clause to filter out values:

```
user> (for [x (range 3 7)]
  (* x x))
(9 16 25 36)
user> (for [x [0 1 2 3 4 5]]
  :let [y (* x 3)]
  :when (even? y)]
  y)
(0 6 12)
```

The `for` macro can also be used over a number of collections, as shown here:

```
user> (for [x ['a 'b 'c]
  y [1 2 3]]
  [x y])
([a 1] [a 2] [a 3] [b 1] [b 2] [b 3] [c 1] [c 2] [c 3])
```

The `doseq` macro has semantics similar to that of `for`, except for the fact that it always returns a `nil` value. This macro simply evaluates the body expression for all of the items in the given bindings. This is useful in forcing evaluation of an expression with side effects for all the items in a given collection:

```
user> (doseq [x (range 3 7)]
  (* x x))
nil
user> (doseq [x (range 3 7)]
  (println (* x x)))
9
16
25
36
nil
As shown in the preceding example, both the first and second `doseq` forms return `nil`. However, the second form prints the value of the expression `(* x x)`, which is a side effect, for all items in the sequence `(range 3 7)

The `into` function can be used to easily convert between types of collections. This function requires two collections to be supplied to it as arguments, and returns the first collection filled with all the items in the second collection. For example, we can convert a sequence of vectors into a map, and vice versa, using the `into` function, shown here:

```Clojure
user> (into {} [[:a 1] [:c 3] [:b 2]])
{:a 1, :c 3, :b 2}
user> (into [] [1 2 3 4])
[[1 2] [3 4]]
```

We should note that the `into` function is essentially a composition of the `reduce` and `conj` functions. As `conj` is used to fill the first collection, the value returned by the `into` function will depend on the type of the first collection. The `into` function will behave similar to `conj` with respect to lists and vectors, shown here:

```Clojure
user> (into [1 2 3] '(4 5 6))
[1 2 3 4 5 6]
user> (into '(1 2 3) '(4 5 6))
(4 5 6 1 2 3)
```

A sequence can be partitioned into smaller ones using the `partition`, `partition-all` and `partition-by` functions. Both the `partition` and `partition-all` functions take two arguments—one for the number of items `n` in the partitioned sequences and another for the sequence to be partitioned. However, the `partition-all` function will also return the items from the sequence, which have not been partitioned as a separate sequence, shown here:

```Clojure
user> (partition 2 (range 11))
((0 1) (2 3) (4 5) (6 7) (8 9))
user> (partition-all 2 (range 11))
((0 1) (2 3) (4 5) (6 7) (8 9) (10))
```

The `partition` and `partition-all` functions also accept a step argument, which defaults to the supplied number of items in the partitioned sequences, shown as follows:

```Clojure
user> (partition 3 2 (range 11))
((0 1 2) (2 3 4) (4 5 6) (6 7 8) (8 9 10))
user> (partition-all 3 2 (range 11))
((0 1 2) (2 3 4) (4 5 6) (6 7 8) (8 9 10) (10))
```
The partition function also takes a second sequence as an optional argument, which is used to pad the sequence to be partitioned in case there are items that are not partitioned. This second sequence has to be supplied after the step argument to the partition function. Note that the padding sequence is only used to create a single partition with the items that have not been partitioned, and the rest of the padding sequence is discarded. Also, the padding sequence is only used if there are any items that have not been partitioned. This can be illustrated in the following example:

```
user> (partition 3 (range 11))
((0 1 2) (3 4 5) (6 7 8))
user> (partition 3 3 (range 11 12) (range 11))
((0 1 2) (3 4 5) (6 7 8) (9 10 11))
user> (partition 3 3 (range 11 15) (range 11))
((0 1 2) (3 4 5) (6 7 8) (9 10 11))
user> (partition 3 4 (range 11 12) (range 11))
((0 1 2) (4 5 6) (8 9 10))
```

In this example, we first provide a padding sequence in the second statement as (range 11 12), which only comprises of a single element. In the next statement, we supply a larger padding sequence, as (range 11 15), but only the first item 11 from the padding sequence is actually used. In the last statement, we also supply a padding sequence but it is never used, as the (range 11) sequence is partitioned into sequences of 3 elements each with a step of 4, which will have no remaining items.

The partition-by function requires a higher-order function to be supplied to it as the first argument, and will partition items in the supplied sequence based on the return value of applying the given function to each element in the sequence. The sequence is essentially partitioned by partition-by whenever the given function returns a new value, as shown here:

```
user> (partition-by #(= 0 %) [-2 -1 0 1 2])
((-2 -1) (0) (1 2))
user> (partition-by identity [-2 -1 0 1 2])
((-2) (-1) (0) (1) (2))
```

In this example, the second statement partitions the given sequence into sequences that each contain a single item as we have used the identity function, which simply returns its argument. For the [-2 -1 0 1 2] sequence, the identity function returns a new value for each item in the sequence and hence the resulting partitioned sequences all have a single element.
The `sort` function can be used to change the ordering of elements in a sequence. The general form of this function requires a function to compare items and a sequence of items to sort. The supplied function defaults to the `compare` function, whose behavior changes depending on the actual type of the items being compared:

```
user> (sort [3 1 2 0])
(0 1 2 3)
user> (sort > [3 1 2 0])
(3 2 1 0)
user> (sort ["Carol" "Alice" "Bob"])
("Alice" "Bob" "Carol")
```

If we intend to apply a particular function to each item in a sequence before performing the comparison in a `sort` form, we should consider using the `sort-by` function for a more concise expression. The `sort-by` function also accepts a function to perform the actual comparison, similar to the `sort` function. The `sort-by` function can be demonstrated as follows:

```
user> (sort #(compare (first %1) (first %2)) [[[1 1] [2 2] [3 3]]])
([[1 1] [2 2] [3 3]])
user> (sort-by first [[[1 1] [2 2] [3 3]]])
([[1 1] [2 2] [3 3]])
user> (sort-by first > [[[1 1] [2 2] [3 3]]])
([[3 3] [2 2] [1 1]])
```

In this example, the first and second statements both compare items after applying the `first` function to each item in the given sequence. The last statement passes the `>` function to the `sort-by` function, which returns the reverse of the sequence returned by the first two statements.

**Filtering sequences**

Sequences can also be filtered, that is transformed by removing some elements from the sequence. There are several standard functions to perform this task. The `keep` function can be used to remove values from a sequence that produces a `nil` value for a given function. The `keep` function requires a function and a sequence to be passed to it. The `keep` function will apply the given function to each item in the sequence and remove all values that produce `nil`, as shown here:

```
user> (keep #(if (odd? %) %) (range 10))
(1 3 5 7 9)
user> (keep seq [[]] '(1 2 3) [:a :b] nil))
((1 2 3) (:a :b))
```
In this example, the first statement removes all even numbers from the given sequence. In the second statement, the `seq` function is used to remove all empty collections from the given sequence.

A map or a set can also be passed as the first argument to the `keep` function since they can be treated as functions, as shown here:

```clojure
user> (keep {:a 1, :b 2, :c 3} [:a :b :d])
(1 2)
user> (keep #{0 1 2 3} #{2 3 4 5})
(3 2)
```

The `filter` function can also be used to remove some elements from a given sequence. The `filter` function expects a predicate function to be passed to it along with the sequence to be filtered. The items for which the predicate function does not return a truthy value are removed from the result. The `filterv` function is identical to the `filter` function, except for the fact that it returns a vector instead of a list:

```clojure
user> (filter even? (range 10))
(0 2 4 6 8)
user> (filterv even? (range 10))
[0 2 4 6 8]
```

Both the `filter` and `keep` functions have similar semantics. However, the primary distinction is that the `filter` function returns a subset of the original elements, whereas `keep` returns a sequence of non-nil values that are returned by the function supplied to it, as shown in the following example:

```clojure
user> (keep #(if (odd? %) %) (range 10))
(1 3 5 7 9)
user> (filter odd? (range 10))
(1 3 5 7 9)
```

Note that in this example, if we passed the `odd?` function to the `keep` form, it would return a list of `true` and `false` values, as these values are returned by the `odd?` function.

Also, a `for` macro with a `:when` clause is translated into an expression that uses the `filter` function, and hence a `for` form can also be used to remove elements from a sequence:

```clojure
user> (for [x (range 10) :when (odd? x)] x)
(1 3 5 7 9)
```
A vector can be sliced using the subvec function. By sliced, we mean to say that a smaller vector is selected from the original vector depending on the values passed to the subvec function. The subvec function takes a vector as its first argument, followed by the index indicating the start of the sliced vector, and finally another optional index that indicates the end of the sliced vector, as shown here:

```
user> (subvec [0 1 2 3 4 5] 3)
[3 4 5]
user> (subvec [0 1 2 3 4 5] 3 5)
[3 4]
```

Maps can be filtered by their keys using the select-keys function. This function requires a map as the first argument and a vector of keys as a second argument to be passed to it. The vector of keys passed to this function indicates the key-value pairs to be included in the resulting map, as shown here:

```
user> (select-keys {:a 1 :b 2} [:a])
{:a 1}
user> (select-keys {:a 1 :b 2 :c 3} [:a :c])
{:c 3, :a 1}
```

Another way to select key-value pairs from a map is to use the find function, as shown here:

```
user> (find {:a 1 :b 2} :a)
[:a 1]
```

take-while and drop-while are analogous to the take and drop functions, and require a predicate to be passed to them, instead of the number of elements to take or drop. The take-while function takes elements as long as the predicate function returns a truthy value, and similarly the drop-while function will drop elements for the same condition:

```
user> (take-while neg? [-2 -1 0 1 2])
(-2 -1)
user> (drop-while neg? [-2 -1 0 1 2])
(0 1 2)
```

**Lazy sequences**

lazy-seq and lazy-cat are the most elementary constructs to create lazy sequences. The value returned by these functions will always have the type clojure.lang.LazySeq. The lazy-seq function is used to wrap a lazily computed expression in a cons form. This means that the rest of the sequence created by the cons form is lazily computed. For example, the lazy-seq function can be used to construct a lazy sequence representing the Fibonacci sequence as shown in Example 1.8:
(defn fibo-cons [a b]
  (cons a (lazy-seq (fibo-cons b (+ a b)))))

Example 1.8: A lazy sequence created using lazy-seq

The `fibo-cons` function requires two initial values, \(a\) and \(b\), to be passed to it as the initial values, and returns a lazy sequence comprising the first value \(a\) and a lazily computed expression that uses the next two values in the sequence, that is, \(b\) and \((+ a b)\). In this case, the `cons` form will return a lazy sequence, which can be handled using the `take` and `last` functions, as shown here:

```
user> (def fibo (fibo-cons 0N 1N))
#'user/fibo
user> (take 2 fibo)
(0N 1N)
user> (take 11 fibo)
(0N 1N 2N 3N 5N 8N 13N 21N 34N 55N)
user> (last (take 10000 fibo))
207936...230626N
```

Note that the `fibo-cons` function from Example 1.8 recursively calls itself without an explicit `recur` form, and yet it does not consume any stack space. This is because the values present in a lazy sequence are not stored in a call stack, and all the values are allocated on the process heap.

Another way to define a lazy Fibonacci sequence is by using the `lazy-cat` function. This function essentially concatenates all the sequences it is supplied in a lazy fashion. For example, consider the definition of the Fibonacci sequence in Example 1.9:

```
(def fibo-seq
  (lazy-cat [0N 1N] (map + fibo-seq (rest fibo-seq))))
```

Example 1.9: A lazy sequence created using lazy-cat

The `fibo-seq` variable from Example 1.9 essentially calculates the Fibonacci sequence using a lazy composition of the `map`, `rest`, and `+` functions. Also, a sequence is required as the initial value, instead of a function as we saw in the definition of `fibo-cons` from Example 1.8. We can use the `nth` function to obtain a number from this sequence as follows:

```
user> (first fibo-seq)
0N
user> (nth fibo-seq 1)
1N
user> (nth fibo-seq 10)
55N
user> (nth fibo-seq 9999)
207936...230626N
```
As shown previously, `fibo-cons` and `fibo-seq` are concise and idiomatic representations of the infinite series of numbers in the Fibonacci sequence. Both of these definitions return identical values and do not cause an error due to stack consumption.

An interesting fact is that most of the standard functions that return sequences, such as `map` and `filter`, are inherently lazy. Any expression that is built using these functions is lazy, and hence never evaluated until needed. For example, consider the following expression that uses the `map` function:

```
user> (def xs (map println (range 3)))
#'user/xs
user> xs
0
1
2
(nil nil nil)
```

In this example, the `println` function is not called when we define the `xs` variable. However, once we try to print it in the REPL, the sequence is evaluated and the numbers are printed out by calling the `println` function. Note that `xs` evaluates to `(nil nil nil)` as the `println` function always returns `nil`.

Sometimes, it is necessary to eagerly evaluate a lazy sequence. The `doall` and `dorun` functions are used for this exact purpose. The `doall` function essentially forces evaluation of a lazy sequence along with any side effects of the evaluation. The value returned by `doall` is a list of all the elements in the given lazy sequence. For example, let's wrap the `map` expression from the previous example in a `doall` form, shown as follows:

```
user> (def xs (doall (map println (range 3))))
0
1
2
#'user/xs
user> xs
(nil nil nil)
```

Now, the numbers are printed out as soon as `xs` is defined, as we force evaluation using the `doall` function. The `dorun` function has similar semantics as the `doall` function, but it always returns `nil`. Hence, we can use the `dorun` function instead of `doall` when we are only interested in the side effects of evaluating the lazy sequence, and not the actual values in it. Another way to call a function with some side effects over all values in a collection is by using the `run!` function, which must be passed a function to call and a collection. The `run!` function always returns `nil`, just like the `dorun` form.
Using zippers

Now that we are well versed with sequences, let’s briefly examine zippers. Zippers are essentially data structures that help in traversing and manipulating trees. In Clojure, any collection that contains nested collections is termed as a tree. A zipper can be thought of as a structure that contains location information about a tree. Zippers are not an extension of trees, but rather can be used to traverse and realize a tree.

The following namespaces must be included in your namespace declaration for the upcoming examples:

```clojure
(ns my-namespace
 (:require [clojure.zip :as z]
 [clojure.xml :as xml]))
```

The following examples can be found in src/m_clj/c1/zippers.clj of the book’s source code.

We can define a simple tree using vector literals, as shown here:

```clojure
(def tree [:a [1 2 3] :b :c])
```

The vector `tree` is a tree, comprised of the nodes `:a`, `[1 2 3]`, `:b`, and `:c`. We can use the `vector-zip` function to create a zipper from the vector `tree` as follows:

```clojure
(def root (z/vector-zip tree))
```

The variable `root` defined previously is a zipper and contains location information for traversing the given tree. Note that the `vector-zip` function is simply a combination of the standard `seq` function and the `seq-zip` function from the `clojure.zip` namespace. Hence, for trees that are represented as sequences, we should use the `seq-zip` function instead. Also, all other functions in the `clojure.zip` namespace expect their first argument to be a zipper.

To traverse the zipper, we must use the `clojure.zip/next` function, which returns the next node in the zipper. We can easily iterate over all the nodes in the zipper using a composition of the `iterate` and `clojure.zip/next` functions, as shown here:

```clojure
user> (def tree-nodes (iterate z/next root))
#'user/tree-nodes
user> (nth tree-nodes 0)
[:a [1 2 3] :b :c] nil
user> (nth tree-nodes 1)
[:a {:l [], :pnodes ...}] nil
user> (nth tree-nodes 2)
[[1 2 3] {:l [:a], :pnodes ...}] nil
user> (nth tree-nodes 3)
[1 {:l [], :pnodes ...}] nil
```
As shown previously, the first node of the zipper represents the original tree itself. Also, the zipper will contain some extra information, other than the value contained in the current node, which is useful in navigating across the given tree. In fact, the return value of the \texttt{next} function is also a zipper. Once we have completely traversed the given tree, a zipper pointing to the root of the tree will be returned by the \texttt{next} function. Note that some information in a zipper has been truncated from the preceding REPL output for the sake of readability.

To navigate to the adjacent nodes in a given zipper, we can use the \texttt{down}, \texttt{up}, \texttt{left}, and \texttt{right} functions. All of these functions return a zipper, as shown here:

\begin{verbatim}
user> (-> root z/down)
[:a {1 [], :pnodes ... }]
user> (-> root z/down z/right)
[[1 2 3] {1 [:a], :pnodes ... }]
user> (-> root z/down z/right z/up)
[[1 2 3] :b :c] nil]
user> (-> root z/down z/right z/right)
[:b {1 [:a [1 2 3]], :pnodes ... }]
user> (-> root z/down z/right z/left)
[:a {1 [], :pnodes ... }]
\end{verbatim}

The \texttt{down}, \texttt{up}, \texttt{left}, and \texttt{right} functions change the location of the root zipper in the \texttt{[:a [1 2 3] :b :c]} tree, as shown in the following illustration:

The preceding diagram shows a zipper at three different locations in the given tree. Initially, the location of the zipper is at the root of the tree, which is the entire vector. The \texttt{down} function moves the location to the first child node in the tree. The \texttt{left} and \texttt{right} functions move the location of the zipper to other nodes at the same level or depth in the tree. The \texttt{up} function moves the zipper to the parent of the node pointed to by the zipper's current location.
To obtain the node representing the current location of a zipper in a tree, we must use the \texttt{node} function, as follows:

\begin{verbatim}
user> (-> root z/down z/right z/right z/node)
:b
user> (-> root z/down z/right z/left z/node)
:a
\end{verbatim}

To navigate to the extreme left or right of a tree, we can use the \texttt{leftmost} and \texttt{rightmost} functions, respectively, as shown here:

\begin{verbatim}
user> (-> root z/down z/rightmost z/node)
:c
user> (-> root z/down z/rightmost z/leftmost z/node)
:a
\end{verbatim}

The \texttt{lefts} and \texttt{rights} functions return the nodes that are present to the left and right, respectively, of a given zipper, as follows:

\begin{verbatim}
user> (-> root z/down z/rights)
([1 2 3] :b :c)
user> (-> root z/down z/lefts)
nil
\end{verbatim}

As the :a node is the leftmost element in the tree, the \texttt{rights} function will return all of the other nodes in the tree when passed a zipper that has :a as the current location. Similarly, the \texttt{lefts} function for the zipper at the :a node will return an empty value, that is \texttt{nil}.

The \texttt{root} function can be used to obtain the root of a given zipper. It will return the original tree used to construct the zipper, as shown here:

\begin{verbatim}
user> (-> root z/down z/right z/root)
[:a [1 2 3] :b :c]
user> (-> root z/down z/right z/root)
[:a [1 2 3] :b :c]
\end{verbatim}

The \texttt{path} function can be used to obtain the path from the root element of a tree to the current location of a given zipper, as shown here:

\begin{verbatim}
user> (def e (-> root z/down z/right z/down))
#'user/e
user> (z/node e)
1
user> (z/path e)
[[[:a [1 2 3] :b :c]
 [1 2 3]]
\end{verbatim}
In the preceding example, the path of the 1 node in a tree is represented by a vector containing the entire tree and the subtree \([1 2 3]\). This means that to get to the 1 node, we must pass through the root and the subtree \([1 2 3]\).

Now that we have covered the basics of navigating across trees, let’s see how we can modify the original tree. The `insert-child` function can be used to insert a given element into a tree as follows:

```clojure
user> (-> root (z/insert-child :d) z/root)
[:d :a [1 2 3] :b :c]
user> (-> root z/down z/right (z/insert-child 0) z/root)
[:a [0 1 2 3] :b :c]
```

We can also remove a node from the zipper using the `remove` function. Also, the `replace` function can be used to replace a given node in a zipper:

```clojure
user> (-> root z/down z/remove z/root)
[1 2 3] :b :c
user> (-> root z/down (z/replace :d) z/root)
[:d [1 2 3] :b :c]
```

One of the most noteworthy examples of tree-like data is XML. Since zippers are great at handling trees, they also allow us to easily traverse and modify XML content. Note that Clojure already provides the `xml-seq` function to convert XML data into a sequence. However, treating an XML document as a sequence has many strange implications.

One of the main disadvantages of using `xml-seq` is that there is no easy way to get to the root of the document from a node if we are iterating over a sequence. Also, `xml-seq` only helps us iterate over the XML content; it doesn’t deal with modifying it. These limitations can be overcome using zippers, as we will see in the upcoming example.

For example, consider the following XML document:

```xml
<countries>
  <country name="England">
    <city>Birmingham</city>
    <city>Leeds</city>
    <city capital="true">London</city>
  </country>
  <country name="Germany">
    <city capital="true">Berlin</city>
    <city>Frankfurt</city>
    <city>Munich</city>
  </country>
</countries>
```
The document shown above contains countries and cities represented as XML nodes. Each country has a number of cities, and a single city as its capital. Some information, such as the name of the country and a flag indicating whether a city is a capital, is encoded in the XML attributes of the nodes.

The following example expects the XML content shown previously to be present in the resources/data/sample.xml file, relative to the root of your Leiningen project.

Let's define a function to find out all the capital cities in the document, as shown in Example 1.10:

```clojure
(defn is-capital-city? [n]
  (and (= (:tag n) :city)
       (= "true" (:capital (:attrs n)))))

(defn find-capitals [file-path]
  (let [xml-root (z/xml-zip (xml/parse file-path))
        xml-seq (iterate z/next (z/next xml-root))
        -> xml-seq
        (take-while #(not= (z/root xml-root) (z/node %)))
        (map z/node)
        (filter is-capital-city?)
        (mapcat :content)))))
```

Example 1.10: Querying XML with zippers

Firstly, we must note that the parse function from the clojure.xml namespace reads an XML document and returns a map representing the document. Each node in this map is another map with the :tag, :attrs, and :content keys associated with the XML node's tag name, attributes, and content respectively.

In Example 1.10, we first define a simple function, is-capital-city?, to determine whether a given XML node has the city tag, represented as :city. The is-capital-city? function also checks whether the XML node contains the capital attribute, represented as :capital. If the value of the capital attribute of a given node is the "true" string, then the is-capital-city? function returns true.
The \texttt{find-capitals} function performs most of the heavy lifting in this example. This function first parses XML documents present at the supplied path \texttt{file-path}, and then converts it into a zipper using the \texttt{xml-zip} function. We then iterate over the zipper using the \texttt{next} function until we arrive back at the root node, which is checked by the \texttt{take-while} function. We then map the \texttt{node} function over the resulting sequence of zippers using the \texttt{map} function, and apply the \texttt{filter} function to find the capital cities among all the nodes. Finally, we use the \texttt{mapcat} function to obtain the XML content of the filtered nodes and flatten the resulting sequence of vectors into a single list.

When supplied a file containing the XML content we described earlier, the \texttt{find-capitals} function returns the names of all capital cities in the document:

\begin{verbatim}
user> (find-capitals "resources/data/sample.xml")
("London" "Berlin" "Paris")
\end{verbatim}

As demonstrated previously, zippers are apt for dealing with trees and hierarchical data such as XML. More generally, sequences are a great abstraction for collections and several forms of data, and Clojure provides us with a huge toolkit for dealing with sequences. There are several more functions that handle sequences in the Clojure language, and you are encouraged to explore them on your own.

\section*{Working with pattern matching}

In this section, we will examine \textit{pattern matching} in Clojure. Typically, functions that use conditional logic can be defined using the \texttt{if, when, or cond} forms. Pattern matching allows us to define such functions by declaring patterns of the literal values of their parameters. While this idea may appear quite rudimentary, it is a very useful and powerful one, as we shall see in the upcoming examples. Pattern matching is also a foundational programming construct in other functional programming languages.

In Clojure, there is no pattern matching support for functions and forms in the core language. However, it is a common notion among Lisp programmers that we can easily modify or extend the language using macros. Clojure takes this approach as well, and thus pattern matching is made possible using the \texttt{match} and \texttt{defun} macros. These macros are implemented in the \texttt{core.match} and \texttt{defun} \texttt{(https://github.com/clojure/core.match)} and \texttt{(https://github.com/killme2008/defun)} community libraries. Both of these libraries are also supported on ClojureScript.
The following library dependencies are required for the upcoming examples:

```clojure
[org.clojure/core.match "0.2.2"
 :exclusions [org.clojure/tools.analyzer.jvm]]
[defun "0.2.0-RC"]
```

Also, the following namespaces must be included in your namespace declaration:

```clojure
(ns my-namespace
 (:require [clojure.core.match :as m]
             [defun :as f]))
```

The following examples can be found in `src/m_clj/c1/match.clj` of the book's source code.

Let's consider a simple example that we can model using pattern matching. The XOR logic function returns a true value only when its arguments are exclusive of each other, that is, when they have differing values. In other words, the XOR function will return false when both of its arguments have the same values. We can easily define such a function using the `match` macro, as shown in *Example 1.11*:

```clojure
(defn xor [x y]
  (m/match [x y]
           [true true] false
           [false true] true
           [true false] true
           [false false] false))
```

*Example 1.11: Pattern matching using the match macro*

The `xor` function from *Example 1.11* simply matches its arguments, `x` and `y`, against a given set of patterns, such as `[true true]` and `[true false]`. If both the arguments are `true` or `false`, then the function returns `false`, or else it returns `true`. It's a concise definition that relies on the values of the supplied arguments, rather than the use of conditional forms such as `if` and `when`. The `xor` function can be defined alternatively, and even more concisely, by the `defun` macro, as shown in *Example 1.12*:

```clojure
(f/defun xor
  ([true true] false)
  ([false true] true)
  ([true false] true)
  ([false false] false))
```

*Example 1.12: Pattern match using the defun macro*
Working with Sequences and Patterns

The definition of the xor function that uses the defun macro simply declares the actual values as its arguments. The expression to be returned is thus determined by the values of its inputs. Note that the defun macro rewrites the definition of the xor function to use the match macro. Hence, all patterns supported by the match macro can also be used with the defun macro. Both the definitions of the xor function, from Example 1.11 and Example 1.12, work as expected, as shown here:

```lisp
user> (xor true true)
false
user> (xor true false)
true
user> (xor false true)
true
user> (xor false false)
false
```

The xor function will throw an exception if we try to pass values that have not been declared as a pattern:

```lisp
user> (xor 0 0)
IllegalArgumentException No matching clause: [0 0] user/xor ...
```

We can define a simple function to compute the $n^{th}$ number of the Fibonacci sequence using the defun macro, as shown in Example 1.13:

```lisp
(f/defun fibo
  (\[0\] 0N)
  (\[1\] 1N)
  (\[n\] (+ (fibo (- n 1))
           (fibo (- n 2)))))
```

**Example 1.13**: A recursive function with pattern matching

Note the use of the variable $n$ in the function's pattern rules. This signifies that any value other than 0 and 1 will match with the pattern definition that uses $n$. The fibo function defined in Example 1.13 does indeed calculate the $n^{th}$ Fibonacci sequence, as shown here:

```lisp
user> (fibo 0)
0N
user> (fibo 1)
1N
user> (fibo 10)
55N
```
However, the definition of \texttt{fibo}, shown in \textit{Example 1.13}, cannot be optimized by tail call elimination. This is due to the fact that the definition of \texttt{fibo} is tree recursive. By this, we mean to say that the expression \((+ (\texttt{fibo} \ldots) (\texttt{fibo} \ldots))\) requires two recursive calls in order to be evaluated completely. In fact, if we replace the recursive calls to the \texttt{fibo} function with \texttt{recur} expressions, the resulting function won’t compile. It is fairly simple to convert tree recursion into linear recursion, as shown in \textit{Example 1.14}:

\begin{verbatim}
(fn/defun fibo-recur
  ([a b 0] a)
  ([a b n] (fibo-recur b (+ a b) (dec n)))
  ([n] (fibo-recur 0N 1N n)))
\end{verbatim}

\textbf{Example 1.14:} A tail recursive function with pattern matching

It is fairly obvious from the definition of the \texttt{fibo-recur} function, from \textit{Example 1.14}, that it is indeed tail recursive. This function does not consume any stack space, and can be safely called with large values of \(n\), as shown here:

\begin{verbatim}
user> (fibo-recur 0)
0N
user> (fibo-recur 1)
1N
user> (fibo-recur 10)
55N
user> (fibo-recur 9999)
207936...230626N
\end{verbatim}

As the preceding examples show us, pattern matching is a powerful tool in functional programming. Functions that are defined using pattern matching are not only correct and expressive, but can also achieve good performance. In this respect, the \texttt{core.match} and \texttt{defun} libraries are indispensible tools in the Clojure ecosystem.

\section*{Summary}

In this chapter, we introduced a few programming constructs that can be used in the Clojure language. We've explored recursion using the \texttt{recur}, \texttt{loop}, and \texttt{trampoline} forms. We've also studied the basics of sequences and laziness, while describing the various functions in the Clojure language that are used in creating, transforming, and filtering sequences. Next, we had a look at zippers, and how they can be used to idiomatically handle trees and hierarchical data such as XML. Finally, we briefly explored pattern matching using the \texttt{core.match} and \texttt{defun} libraries.

In the next chapter, we will explore concurrency and parallelism. We will study the various data structures and functions that allow us to leverage these concepts in Clojure in ample detail.
Where to buy this book

You can buy Mastering Clojure from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.