What this book will do for you...

- Explore alternative uses of classical data structures such as arrays and linked lists
- Explore advanced machine learning and optimization techniques
- Utilize the Clojure libraries, such as Instaparse for parsing, core.match for pattern matching, clojure.zip for zippers, and clojure.matrix for matrix operations
- Learn logic programming through the core.logic library
- Master asynchronous programming using the core.async library
- Observe transducers while resolving real-world use cases

Inside the Cookbook...

- A straightforward and easy-to-follow format
- A selection of the most important tasks and problems
- Carefully organized instructions to solve problems efficiently
- Clear explanations of what you did
- Solutions that can be applied to solve real-world problems

Clojure Data Structures and Algorithms Cookbook

25 recipes to deeply understand and implement advanced algorithms in Clojure

Rafik Naccache


$ 49.99 US
£ 31.99 UK


Prices do not include local sales tax or VAT where applicable

7.50 x 9.25
235 mm x 191 mm

456 pages
11.58mm

Content Type: Black & White
Paper Type: White
Page Count: 216
File Type: InDesign
Request ID: CSS1444725
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Revisiting Arrays'
- A synopsis of the book’s content
- More information on Clojure Data Structures and Algorithms Cookbook
Rafik Naccache is a Tunisian, who is experienced in software architecture and is an emergent technology enthusiast. He earned his bachelor's degree in computer science engineering from the University of Tunis in 2001. Rafik fell in love with Clojure back in 2012 and has been developing it professionally since 2013. He has occupied various positions in the telecom and banking sectors and has launched a few innovative start-ups on the Internet, in which he was able to deploy Clojure apps. He also founded the Tunisian Clojure enthusiasts community. He contributes to open source projects such as Cryogen (https://github.com/cryogen-project/cryogen/graphs/contributors) and Milestones (https://github.com/automagictools/milestones).
Preface

The invention of Lisp by John McCarthy in 1958 is certainly one of the most seminal events in the history of computer science. Originally intended as a way of porting Alonzo Church's lambda calculus theory into the realm of computer programs, Lisp pioneered many original ideas, such as higher order functions, recursion, and even garbage collection that proved to be so highly pragmatic that practically every modern high-level programming language of today is very likely built on some significant Lisp legacy.

However, beyond any practical or technical contribution, Lisp's most important trait is undeniably its unparalleled expressiveness. It's simplistic, yet its extremely elegant syntax propels it as a privileged tool for creative computer scientists, one which you could use as a powerful "building material" to erect algorithmic monuments without worrying about ancillary implementation details. It's certainly this ability of abstracting away "incidental complexity" that made Lisp the language for conducting Artificial Intelligence experiments, for instance.

Clojure, as a modern Lisp language, leverages this extraordinary expressive power to provide a platform that is highly suitable for algorithmic exploration. The abstractions it offers, the functional approach it suggests, as well as the built-in concurrency it supports, are all valuable facilities that enable straight and noise-free problem solving, which is an alternative way of approaching algorithms' design and sometimes, even innovative out-of-the-box thinking.

This book tries to underpin this idea. Using Clojure, we'll consider seven areas of algorithmic challenges and try to address them by taking advantage of all the power that we can get from this Lisp dialect. Besides, while choosing these seven problem domains, I tried to attain two main objectives. First, I wanted to tackle algorithms that have concrete real-world usage, and theory in the recipes will only be present to serve well-defined use cases in our everyday work with computers. Second, I tried to come up with material as varied as possible, so that the recipes cover a wide range of topics, from compressing files and building parsers to designing HTML5 interactive games and assembling type inferencers.
As far as the recipes' general layout is concerned, at the beginning, you will be given a thorough introduction to the intuition and the theory behind the algorithm being studied. Then, I will elaborate on every recipe's detail, making sure that I've explained every step and extensively commented the code. At the end of every recipe, you will see a sample usage of what has been implemented. This way, you will be guided through the whole process: the algorithm inception, its implementation, and its testing. In this process, I hope to have had mirrored the Clojure interactive workflow, in which the developer builds their program function by function, going back and forth to his/her REPL.

I really enjoyed the process of writing this book. I can, for sure, assert that the Clojure promise of high expressive power has been fulfilled for this particular project, as I came up with quite complex algorithmic implementations with reasonable effort while being really productive. My only wish is that you, by the end of this work, will be as convinced by Clojure and Lisp as I am.

You can reach me at @turbopape on Twitter, I'll answer, with extreme pleasure, all your questions.

What this book covers

Chapter 1, Revisiting Arrays, explores some interesting alternative uses for the array data structure. You'll learn in this recipe how to implement data compression using the LZ77 algorithm. Then, we'll see how you can use Pascal's triangle in order to draw some fractals. Next, we'll design a little multithreaded program execution simulator. We will end this chapter by studying an algorithm used to handle a call stack frames operation during a program execution.

Chapter 2, Alternative Linked Lists, delves into the advanced matters related to linked lists. We will cover a method using XOR addressing in order to get doubly linked lists. We'll then cover how to speed up a linked list's element retrieval, thanks to caching. We'll also use this data structure to build a shift-reduce parser. At the end, we'll explore an immutable functional data representation of linked lists by using the skew binary numbers representation.

Chapter 3, Walking Down Forests of Data, focuses on recipes related to tree data structure. First, we'll cover the self-balancing, search-optimized splay tree. Then, we'll elaborate on B-trees and show you how we can use a B-tree in order to build a key-value data store. Next, we'll show you how ropes can be used in order to create an undo-capable text editor. The last recipe of this chapter will be about tries and how they allow you to create efficient autocomplete engines.
Chapter 4, Making Decisions with the Help of Science, gives you an overview of a few machine learning and optimization algorithms. We'll first show you an approach that is used to build live recommendation engines. Then, we'll use the branch and bound algorithm to solve a cost/profit optimization problem, which can only accept a natural numbers solution. Next, we'll use the Dijkstra algorithm in order to find the optimal paths in graphs. The final recipe in this chapter is about using the LexRank algorithm in order to summarize text documents.

Chapter 5, Programming with Logic, focuses on logic programming. We'll first use this highly declarative approach in order to draw interesting facts out of a social networking website's traffic data. Then, we'll show you how a simple type inferencer can be built using logic programming. At the end, we'll design a simple IA module capable of playing one round of checkers.

Chapter 6, Sharing by Communicating, gives particular attention to asynchronous programming. We'll begin by using this concurrency paradigm in order to build a tiny web scraper. Then, we'll go through the process of creating an interactive HTML5 game. Finally, we'll design an online taxi-booking platform as a complex system that could benefit from asynchronous programming.

Chapter 7, Transformations as First-class Citizens, dives into a few particular algorithmic cases inherent to the functional nature of Clojure. We'll start by designing a simple recursive descent parser, making use of the efficient mutual recursion offered by the Trampoline construct. Then, we'll see the new Clojure 1.7 feature—the transducers—in action while developing a mini firewall simulator. Finally, we'll introduce you to the continuation-passing style while designing a little symbolic expression unification engine.
Revisiting Arrays

In this chapter, we will see how you can use array abstractions in Clojure to cover the following topics:

- Efficiently compressing a byte array
- Using Pascal's triangle to draw fractals
- Simulating multithreading using time-sharing
- Simulating a call stack using arrays

Introduction

In this book, we will go on a journey through the broad land of algorithms and data structures, taking a ride on the comfortable vehicle that is Clojure programming language.

First, we will take a look at arrays, exploring their particular structures to tackle problems as interesting as compression, fractal drawing, multithreading, and call stacks.

Then we will elaborate on linked lists, transforming them into doubly linked lists. We will do this to speed up access to their elements, build parsers, and devise fast random access.

The next step of our trip will concern trees of data. We'll show you how to implement self-balancing red-black trees, how to design efficient key-value stores — thanks to B-trees (way to go in order to design undo-capable text editors), and lastly, a methodology to construct autocomplete text typing systems.

After that, we'll focus on exploring some optimization and machine-learning techniques. We will see how to set up a recommendation engine, the way to go to optimize a problem where costs and profits are involved, a methodology to find the best possible paths in graphs, and how to summarize texts.
Then we'll study the topic of logic programming, analyzing some website traffic logs to detect visitors of interest to us. Doing this, we'll dive into the problem of type inferencing for the Java language, and simulate a turn of a checkers game.

At that point, we'll talk about asynchronous programming as a means of tackling difficult problems. We'll build a tiny web spider, design an interactive HTML5 game, and design a complex online taxi-booking solution.

The last rally point of this trip, but certainly not the least, is that we'll have a look at the higher order functions and transducers at the heart of Clojure. We'll design a recursive descent parser using a trampoline, build a reusable mini firewall thanks to transducers, and lastly, explore the continuation passing style as a tool to design a simple unification engine.

This will be quite a tour, in which we will bring to life various real-world use cases related to the essential theory of computing as far as data structures and algorithms are involved, which are all served by the high expressive power of Clojure. By the end of this book, you'll be familiar with many of the advanced concepts that fuel most of the nontrivial applications of our world while you enhance your mastery of Clojure!

## Efficiently compressing a byte array

Compressing a byte array is a matter of recognizing repeating patterns within a byte sequence and devising a method that can represent the same underlying information to take advantage of these discovered repetitions.

To get a rough idea of how this works, imagine having a sequence as:

```plaintext
["a" "a" "a" "b" "b" "b" "b" "b" "b" "c" "c"]
```

It is intuitively more efficient to represent this as:

```plaintext
[3 times "a", 7 times "b", 2 times "c"]
```

Now, we are going to use a methodology based on a well-known algorithm, that is, the LZ77 compression method. LZ77 is, despite being quite old, the basis of most of all the well-known and currently used compression methods, especially the Deflate algorithm.

Deflate is at the heart of the ZIP family of compression algorithms. It uses a slightly modified version of LZ77 plus a special encoding, that is, the Huffman encoding.

The point of LZ77 is to walk a sequence and recognize a pattern in the past elements that will occur in the upcoming elements, replacing those with a couple of values: how many elements should go backwards in order to locate the recurring pattern, which is called "distance"; and how long the recurring pattern, which is labeled as "length".
The iteration of the LZ77 compression would look as follows:

1. At any point of time, the algorithm is processing a particular element, which is located at the current position. Consider a window of the size \( n \), as a set of \( n \) elements preceding the one that is occupying current position, and consider lookahead as the rest of the elements up until the input's end.

2. Begin with the first element of the input.

3. Move on to the next element.

4. Find in the window (that is, past \( n \) elements), the longest pattern that can be found in lookahead.

5. If such a sequence is found, consider distance as the location where, the matching sequence was found, expressed in regards to the current position, consider length as the length of the matching pattern, and proceed with the two following actions:
   - Replace the match in lookahead by "distance" and "length".
   - Move forward using the "length" elements and resume algorithm execution at step 4.

6. Otherwise, resume at step 3.

The procedure to uncompress is as follows:

1. Walk the compressed sequence.

2. If the "distance" and "length" are found, go back to the "distance" elements and replace this couple with the "length" elements.

3. If not, lay out the element that you've found.

Let's see this in action in Clojure!

**How to do it...**

1. First of all, here is the \( \text{ns} \) declaration containing the Clojure facilities that we are going to use:

   ```clojure
   (ns recipe1.core
     (:require [clojure.set :as cset]))
   ;; => we'll need set operations later on.
   ```

2. Let's begin by working on the uncompressing part. First of all, we need an `expand` function that takes the source array as a vector of the elements distance and length and generates a repetition of a sub-vector of the last distance characters from the source array until the length is reached:

   ```clojure
   (defn expand
     [the-vector
      distance
      ```
Revisiting Arrays

length]
(let [end (count the-vector)
    start (- end
distance)
;;=> Here we go backwards 'distance' elements.
pattern (subvec the-vector
    start
    end)] ;;=> We have our pattern.
    (into [] (take length ;;=> We exactly take "length" from
        (cycle pattern))))
;; an infinite repetition of our pattern.

3. Now, let's define un-LZ77 using expand function while walking through a sequence of bytes:

(defn un-LZ77
    [bytes]
    (loop [result []
            remaining bytes]
        ;;=> We recur over the contents of the array.
        (if (seq remaining)
            (let [current (first remaining)
                the-rest (rest remaining)]
                ;;=> Current element under scrutiny;
                (if-not (vector? Current)
                    ;;=> If it is not a vector, add to result
                    (recur (conj result
                        ;; the very element, and move on.
                        current)
                        the-rest)
                    (recur (into result (expand result
                        ;;=> This is a vector, then we'll expand here and move on
                        (current 0)
                        (current 1)))
                        the-rest))))
        ;;=> end of recursion, return result.
4. Now let's address the topic of compressing. First of all, we need to grab all sub-vectors, as we'll have to find matches between window and lookahead and then pick the longest one among them:

```clojure
(defn all-subvecs-from-beginning
  ;; => this function will generate a set of all sub-vectors starting
  ;; from begin
  [v]
  (set (map #(subvec v 0 %)
             ;; => we apply subvec from 0 to all indices from 1 up to the size
             ;; of the array + 1.
             (range 1 (inc (count v))))))

(defn all-subvecs
  ;; => this function will generate all
  ;; [v]          ;       sub-vectors, applying
  (loop [result #{},
          ;       all-subvecs from beginning to
          ;       all possible beginnings.
          remaining v]
    (if (seq remaining)
      (recur (into result
                 (all-subvecs-from-beginning remaining))
          (into[] (rest remaining)))
    ;; => We recur fetching all sub-vectors for next beginning.
      result))
  ;; => end of recursion, I return result.

5. Now we define a function to grab the longest match in left array with the beginning of right array:

```clojure
(defn longest-match-w-beginning
  [left-array right-array]
  (let [all-left-chunks (all-subvecs left-array)
         all-right-chunks-from-beginning
         ;; => I take all sub-vectors from left-array
         all-subvecs-from-beginning right-array)
    ;; => I take all sub-vectors from right-array
```
Revisiting Arrays

all-matches (cset/intersection all-right-chunks-from-beginning
          all-left-chunks)]

;;=> I get all the matchings using intersection on sets
  (-=> all-matches
    (sort-by count >)
    first)))

;;=> Then I return the longest match, sorting them
;_; by decreasing order and taking the first element.

6. With the longest match function in hand, we need a function to tell us where
   this match exactly located inside the window:
(defn pos-of-subvec
  [sv v]
  {:pre \[(<= (count sv)
            (count v))]}

;;=> I verify that sv elements are less than v's.
  (loop
    [cursor 0]
    (if (or (empty? v)
             (empty? sv))
        ;;=> If on of the vectors is empty
        (= cursor (count v)))
    ;; or the cursor ended-up exiting v,
    nil
    ;; we return nil
    (if (= (subvec v cursor
             (+ (count sv)
                ;; beginning with cursor up to sv count
                cursor)) sv)
      ;; is equal to sv cursor
      ;_; we return cursor, this is where the match is.
      (recur (inc cursor)))
    ;;=> We recur incrementing the cursor)
7. Armed with the toolbox we’ve built so far, let’s devise an LZ77 step:

(defn LZ77-STEP
  [window look-ahead]
  (let [longest (longest-match-w-beginning window look-ahead)] ;;=> We find the Longest match,
    (if-let [pos-subv-w (pos-of-subvec longest window)] ;;=> If there is a match we find its position in window.
      (let [distance (- (count window) pos-subv-w) ;;=> the distance,
        pos-subv-l (pos-of-subvec longest look-ahead) ;;=> the position of the match in look-ahead
        the-char (first (subvec look-ahead (+ pos-subv-l (count longest))))) ;;=> the first element occurring after the match
        {:distance distance
         :length (count longest)
         :char the-char}) ;;=> and we return information about match
      {:distance 0
       :length 0
       :char (first look-ahead)}) ;;=> We did not find a match, we emit zeros for "distance"
    ;; and "length", and first element of lookahead as first char
    ;; occurring after the (non-)match.
  )

8. Finally, we will write the main LZ77 compression function as follows:

(defn LZ77
  [bytes-array window-size]
  (->> (loop [result []
              cursor 0
              window []
              ]
    ...
look-ahead bytes-array]
;;=> we begin with position 0; and everything as look-ahead.
(if (empty? look-ahead)
  result
;;=> end of recursion, I emit result.
  (let [this-step-output (LZ77-STEP window look-ahead)
    distance (:distance this-step-output)
    length (:length this-step-output)
    literal (:char this-step-output)
    ;;=> We grab informations about this step output
    raw-new-cursor (+ cursor
      length
      1)
    new-cursor (min raw-new-cursor
                     (count bytes-array))
    ;;=> We compute the new-cursor, that is, where to go in the next
    ;; step
    ;; which is capped by count of bytes-array
    new-window (subvec bytes-array
                 (max 0 (inc (- new-cursor
                              window-size)))
                 new-cursor)
    ;;=> new window is window-size elements back from new cursor.
    new-look-ahead (subvec bytes-array
                     new-cursor ])
    ;;=> new look-ahead is everything from new cursor on.
    (recur (conj result
               [distance length]
               literal)
         new-cursor
         new-window
         new-look-ahead)))
;; and we recur with the new elements.
(filter   (partial
    not=
(filter (comp not nil?)) ;; => and any nils
(into [])) ;; => and make a vector out of the output.

That's it! Now, let's see our code in action. Input into your REPL as follows:

(LZ77 ["a" "b" "c" "f" "a" "b" "c" "d"] 5)
;; => ["a" "b" "c" "f" [4 3] "d"]
(un-LZ77 ["a" "b" "c" "f" [4 3] "d"])
;; => ["a" "b" "c" "f" "a" "b" "c" "d"]

Using Pascal's triangle to draw fractals

Triangles are a particular matrix type. Each line contains exactly as many nonzero elements as the line index in the matrix. Here is a sample triangle depicted as a vector of vectors in Clojure:

\[
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1
\end{bmatrix}
\]

Now, we can simply omit the zeros altogether and get a real triangle, graphically speaking:

\[
\begin{bmatrix}
1 \\
1 & 1 \\
1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1
\end{bmatrix}
\]
Pascal's triangle is a matrix whose elements are computed as the sum of the elements that are directly above it and the element to the left of the elements that are directly above it. The very first element is 1. This matrix was devised by Pascal as a means of computing the powers of binomials. Here's a Pascal's triangle for up to seven lines:

\[
\begin{bmatrix}
1 \\
1 & 1 \\
1 & 2 & 1 \\
1 & 3 & 3 & 1 \\
1 & 4 & 6 & 4 & 1 \\
1 & 5 & 10 & 10 & 5 & 1 \\
1 & 6 & 15 & 20 & 15 & 6 & 1
\end{bmatrix}
\]

If we look at this Pascal's triangle, then a binomial, let's say \((a+b)\), elevated to the power 4 is computed by extracting the coefficient from the row with index 4 (first row is having index 0). The resulting polynomial is: \(a^4b+4a^3b+6a^2b^2+4ab^3+ab^4\).

Now, it happens that plotting odd elements from a Pascal's triangle yields a fractal, that is, an image that infinitely repeats itself.

Such a fractal derived from plotting the odd elements of a Pascal's triangle is known as the Sierpinski triangle.

If you closely watch the triangle's structure, you'll notice that each line is symmetrical. As such, for the sake of efficiency, you only have to compute half of a line at a time and append it to its own mirrored copy to get the whole line.

Besides, as our main purpose is to draw fractals, we'll have to generate a huge Pascal's triangle, in order to have a proper image. Doing so will make us soon hit number limitations and we'll have to circumvent this. Luckily, summing the remainder of a division of two by two numbers leads to the same even properties, as if you've summed those very numbers. Then, our implementation will rely on this to come up with sufficiently big images; we'll create Pascal's triangles with the sums of the remainder of the division by two.

How to do it...

1. First of all we'll need to import, along with our ns declaration, some Java facilities to help us build the fractal and write it to a file:

   \[
   \text{(ns recipe2.core}
   \text{   (:import (java.awt image.BufferedImage Color)}
   \text{   ;=> so we can plot.}
   \text{   (javax.imageio ImageIO)}
   \text{   (java.io File))) ;=> so we can write to a file.}
   \]
2. Let's write a function to compute a particular row in a Pascal's triangle. As we've discussed, in a Pascal's triangle you compute a row of a particular index based on the one located directly above it (of the preceding index), that's why this function takes one row as input. Here we pass a yield function, permitting it to compute an element out of its immediately preceding neighbor and the element to the left of the preceding neighbor. Each time, we compute half a line and append it to its reverse:

```clojure
(defn pascal-row-step
    [yield pascal-row]
    {:pre [(> (get pascal-row 0) 0)]}  ;=> We can only start from [1]!
    (let [cnt-elts (count pascal-row)
           half-row (subvec pascal-row 0
                       (inc (double (/ cnt-elts 2))))]
      ;;=> We compute half the above row
      padded-half-row (into [0] half-row)
      ;;=> and add a 0 to the beginning, as we'll use it in computation
      half-step (vec (map (comp (partial apply yield)
                              vec)
                       (partition 2 1
                        padded-half-row)))
      ;;=> we compute the first half, summing the above element
      ;;      and the element at the left of the above one.
      other-half-step (vec (if (even? cnt-elts)
                             (-> half-step
                                butlast
                                reverse)
                             (-> half-step
                                reverse)))
      ;;=> the mirror of the half we computed. If count elements is
      ;; even, we omit the last element from half-step.
      (into half-step other-half-step))
      ;;=> we return half to which we append the mirror copy.

3. Now, we'll build the whole Pascal's triangle parameterized with the yield function:

```clojure
(defn pascal-rows
    [yield row-number]
    (loop [nb 0
           result []
           latest-result [1]]
      ;;=> We'll loop using pascal-row-step,
      ;;=> keeping track of the last
      ;;computed line at each step of the recursion.
      )
```
(if (<= nb row-number)
  ;;=> the counter did not still reach the end
  (recur (inc nb)
    (conj result latest-result)
    (pascal-row-step yield latest-result))
  ;;=> We recur incrementing the counter, feeding the new line to
  ;; result and keeping track of the last computed line.
  result))

;;=> end of the recursion, emitting result.

4. We will also prepare a yield function to compute the remainder of the sum of two
numbers:
(defn even-odd-yield
  [n1 n2]
  (mod (+ n1 n2) 2))

5. We will prepare a helper function to generate the fractals:
(def gr-triangles (partial pascal-rows even-odd-yield))

6. Now we can just launch the following to have our graphical 0-1 fractal representation:
(gr-triangles 10)

7. With gr-triangles under our belt, we have to plot points at the positions that hold
1. For this, we'll consider the cords of such positions to be the index of line and the
index of elements in the vector held by this line that have the value 1:
(defn draw [size]
  (let [img (BufferedImage. size size BufferedImage/TYPE_INT_ARGB)
    ;;=> Creating img as a Buffered Image
    plot-rows (gr-triangles size)
    ;;=> computing the triangle of 0 and 1
    plots (for [x (range 0 size) y (range 0 x)]
      (if (= 1 (get (get plot-rows x) y))
        [x y]))
    ;;=> we save the positions holding 1 in vectors. As the structure
    ;; is triangular;
    ;; the first counter, "x" goes up to "size", and the second one,
    ;; "y",
    ;; goes up to "x"
gfx (.getGraphics img)]

;; => we get the graphics component, where to draw from the Java Object.
(.setColor gfx Color/WHITE)
(.fillRect gfx 0 0 size size )

;; => we set a white background for the image.
(.setColor gfx Color/BLACK)

;; => We set the pen color to black again
(doseq [p (filter (comp not nil?) plots)]
  (.drawLine gfx
    (get p 0)
    (get p 1)
    (get p 0)
    (get p 1)))

;; => We plot, by drawing a line from and to the same point.
(ImageIO/write img "png"
  (File. "~/your/location/result.png")))

;; => and we save the image as a png in this location.
;; Be sure to set a correct one when running on your machine!

Here is a zoomed-out image generated by this function of the size 10,000:
Revisiting Arrays

Here is a zoomed-in view of some parts of it:

![Image of triangles]

Here, the same triangles appear over and over again as you zoom in on the image.

Simulating multithreading using time-sharing

Time-sharing is about sharing a computing facility between multiple concurrent processes. At its very basic version, a scheduler decides, which one of these competing processes to execute at every single quantum of time. This way, even a single processor core, only capable of sequential operation, is able to spawn multiple threads, as if they were being executed in parallel.

One method of preventing race conditions, that is, multiple processes concurrently reading and writing wrong versions of the same shared place in memory, is locking. Imagine, for example, that there are two processes incrementing the same shared counter. Process 1 takes the value of the counter and overwrites it with the value $+1$. If, meanwhile, process 2 does the same thing – that is, it reads the same version of the counter that process 1 reads for the very first time and overwrites it with the same value $+1$ – you'd end up with the counter that will only be incremented once. Hence, locking this portion of code makes process 2 wait for process 1 until it finishes reading and writing, and only when process 1 is done and sets the lock free, process 2 will be allowed to play its part, leading to the correct final value of the counter $+2$. 
Managing locks can be too tedious. That’s why it is often better to use high-level concurrency alternatives, such as those provided by Clojure: the software transactional memory (refs and atoms), agents, and core.async.

How to do it...

1. First of all, we’ll begin importing some libraries that we will use:

   (ns recipe3.core
     (:require [instaparse.core :as insta]
               ;=> For parsing the code of our processes
       ;processes
               [:require [clojure.zip :as z]]
               ;=> To walk the parse-trees and generate processes instructions.
       ;require [clojure.pprint :refer :all]])
     ;=> an Alias to easily pretty print our outputs.

   Instaparse (https://github.com/engelberg/instaparse) is a parser generator written in Clojure. To explain all of the mechanism behind Instaparse is beyond the scope of this book, but you should know that it handles context-free grammar (CFG) and generates parse trees of your input programs according to these grammar concepts.

2. To be able to pretty-print our output in the REPL, let’s define an alias for clojure.pprint/pprint, so that we can make it more conveniently:

   (def p pprint)

3. As we’ll be spawning processes with instructions of their own, let’s define a minimal language that instaparse will be able to interpret for us. Our language instructions for a single process are as follows:

   heavy-op op1;
   light-op op2;
   lock l1;
   medium-op op3;
   unlock l1;
4. The previous snippet is self-explanatory. Our language only contains three types of operations: heavy-op, which are sorted according to the effort they need in order to be fully processed by the scheduler: heavy-op, medium-op, and finally light-op. Besides, we are able to lock and unlock a portion of our programs with the lock and unlock instructions. Each one of these instructions needs you to specify an identifier, so that they can be recognized in the scheduler output.

5. The grammar for such a language is:

```lisp
def r3-language
"S = INSTRS
INSTRS = ((INSTR | LOCKED-INSTRS) <optional-whitespace>)*
INSTR = HEAVY-OP | MEDIUM-OP | LIGHT-OP
HEAVY-OP = <optional-whitespace> 'heavy-op' <whitespace> ID <SEP>
MEDIUM-OP = <optional-whitespace> 'medium-op' <whitespace> ID <SEP>
LIGHT-OP = <optional-whitespace> 'light-op' <whitespace> ID <SEP>
LOCKED-INSTRS = LOCK INSTRS UNLOCK
LOCK = <optional-whitespace> 'lock' <whitespace> ID <SEP>
UNLOCK = <optional-whitespace> 'unlock' <whitespace> ID <SEP>
ID = #'[a-zA-Z0-9]+'
PRIORITY = #'[0-9]+'
whitespace = #'\s+'
optional-whitespace = #'\s*'\s+
SEP = #'\s* ';'\n```

6. Note that identifiers between angle brackets will not be seen in the parse tree, so there's no use referring to the white-space tags, for instance.

7. Let's see what would be the Instaparse output for the program we wrote in the preceding code. For this, just type the following in your REPL:

```lisp
(p (insta/parse (insta/parser r3-language)
"heavy-op op1;
light-op op2;
lock l1;
medium-op op3;
unlock l1;")
And you'll get :
[:S
 [:INSTR [:HEAVY-OP "heavy-op" [:ID "op1"]]]
 [:INSTR [:LIGHT-OP "light-op" [:ID "op2"]]]
```
8. We need to transform these nested vectors into instructions. First of all, we will make use of the very handy instaparse function transform to eliminate the rules tags and get a more useful representation of our instructions. transform function takes a tag and applies a function to the elements next to it in the vector that this tag refers to:

```clojure
(defn gen-program
  [parser program]
  (insta/transform
   {:S identity
    :INSTRS (fn [& args] (vec args))
    :INSTR identity
    :HEAVY-OP (fn [x y] {:inst-type :heavy-op :inst-id (get y 1) :inst-op op1})
    :MEDIUM-OP (fn [x y] {:inst-type :medium-op :inst-id (get y 1) :inst-op op3})
    :LIGHT-OP (fn [x y] {:inst-type :light-op :inst-id (get y 1) :inst-op op2})
    :LOCKED-INSTRS (fn [& args] (vec args))
    :LOCK (fn [x y] {:inst-type :lock :inst-id {:lock l1}})
    :UNLOCK (fn [x y] {:inst-type :unlock :inst-id {:unlock l1}})
   );;=> This map tells 'transform' how to transform elements next to each tag.
   (parser program))

;; The raw parse tree emitted by Insaparse.
9. Here is the output of gen-program. Input the following code in the REPL:

```clojure
(p (gen-program (insta/parser r3-language)
  "heavy-op op1;
  light-op op2;
  lock l1;
  medium-op op3;
  unlock l1;")
```

10. You'll get the following output:

```clojure
[{:inst-type :heavy-op, :inst-id "op1"}
 {:inst-type :light-op, :inst-id "op2"}
{:inst-type :lock, :inst-id {:lock "l1"}}
{:inst-type :medium-op, :inst-id "op3"}
{:inst-type :unlock, :inst-id {:unlock "l1"}}]
```
Revisiting Arrays

11. To get rid of the nesting that we still see here, we are going to use a zipper, which is a Clojure facility to walk trees. Basically, we will loop all the nested vector elements and only take maps, so that we end up with a nice, flat program structure. As this will be our actual process, we’ll also append a process-id attribute and a priority attribute to its output:

```clojure
(defn fire-a-process
  [grammar
   program
   process-id
   priority]
  (let [prsr (insta/parser grammar) ;=> the parser
        vec.instructions (gen-program prsr program)
        ;=> the nested structure
        zpr (z/vector-zip vec.instructions)]
    (loop [result [],
           loc (-> zpr z/down)]
      (if (z/end? loc)
        ;=> the end of recursion, no more nodes to visit
        {:process-id process-id
         :instructions result
         :priority priority};=> We generate the process
        (recur (if (map? (z/node loc))
                 (conj result (z/node loc))
                 result)
        );=> We only append to result the elements of type 'map'
        (z/next loc)))))

12. Here is a process spawned by our program named :process-1 that has the priority 10. Input the following in your REPL:

```clojure
(fire-a-process r3-language
    "heavy-op op1;
     light-op op2;
     lock l1;
     medium-op op3;
     unlock l1;"
    :process-1
    10)
13. You'll get the following output:
{:process-id :process-1,
 :instructions
 [{:inst-type :heavy-op, :inst-id "op1"}
  {:inst-type :light-op, :inst-id "op2"}
  {:inst-type :lock, :inst-id {:lock "l1"}}
  ;;=> note that ':inst-id' of locks are {':lock' or ':unlock' id},
  ;; so a locking and an un-locking instructions are not mistaken
  ;; one for another.
  {:inst-type :medium-op, :inst-id "op3"}
  {:inst-type :unlock, :inst-id {:unlock "l1"}}],
 :priority 10}

14. Now, we need to set effort for each of our instructions, that is, how many processor
cycles each one of them takes to be executed:
(def insts-effort {:heavy-op 10 :medium-op 5 :light-op 2 :lock 1 :unlock 1})

15. Now we'll concern ourselves with locking. First of all, we need to
find the indices of
locking instructions in our instructions vector:
(defn all-locks-indices [instructions]
  ;;=> 'instructions' is the ':instructions vector' of the output of
  ;; fire-process.
  (let [locks (filter #(= (:inst-type %) :lock)
           instructions)
    lock-indices (map (fn [l] {:lock-id (l :inst-id)
                               :lock-idx (.indexOf
                               instructions l)})
                   locks)]
    ;;=> output of this is : ({:lock-id {:lock "l1"}, :lock-idx 2})

16. With our locks recognized, we can tell which lock every instruction depends on. This is
basically done by finding out which locks have indices inferior to the instruction index:
(defn the-locks-inst-depends-on [instructions instruction]
  (let [the-inst-idx (.indexOf instructions instruction)
        the-lock-idxs (all-locks-indices instructions)]
    (into [] (->> the-lock-idxs
                (filter #(> the-inst-idx (:lock-idx %) ))
                (map :lock-id)))))
Revisiting Arrays

17. We’ll need a map that maintains the state of locks so the scheduler can track the locking and unlocking activities during the program execution with. We’ll define lock and un-lock functions to do this:

```clojure
(defn lock
  "locks lock lock-id in locks map"
  [locks process-id lock-id]
  (assoc locks lock-id {:locker process-id :locked true}))
(defn unlock
  "unlocks lock lock-id in locks map"
  [locks process-id lock-id]
  (assoc locks lock-id {:locker process-id :locked false}))

;;=> The locks state contains its locked state and which process did lock it.

18. The locker process information, manipulated in the previous step is important. As some process’ instruction can only be denied access to a shared resource by locks set by other processes contains, we need to track which is locking what. The is-locked? function relies on this mechanism to inform whether an instruction is locked at any point in time, so it cannot be fired by the scheduler:

```clojure
(defn is-locked?
  [process-id instructions locks instruction]
  (let [inst-locks (the-locks-inst-depends-on instructions instruction)]
    (some true? (map #(and (not= process-id ((get locks %) :locker))
                           ((get locks %) :locked))
                     inst-locks)))

;;=> If some of the locks the instruction depend on are locked (:locked true)
;; and the locker is not its process, then it is considered as locked.

19. Let’s focus on the scheduler now. Imagine that some parts of a process have already been assigned some quanta of time. We need a map to maintain a state for all the processes regarding the parts that already have been processed so far. We’ll call this map scheduled. Let’s say that this map should look like the following:

```
[{:process-id :process-1
  :instructions
  [{:times [1 2 3], :inst-id "op1", :inst-type :heavy-op}]
```

---

20
20. We'll prepare a helper function, scheduled-processes-parts, that'll count the number of quanta already allocated, and this will be handy in knowing whether an instruction is complete:

```clojure
(defn scheduled-processes-parts
  [scheduled]
  (into [] (map (fn [p] {:process-id (:process-id p)
                        :instructions (into []
                                       (map (fn [i] {:inst-id (:inst-id i)
                                                     :inst-type (:inst-type i)
                                                     :count (count (:times i))})
                                       (:instructions p))))
            scheduled)))
```

=> this functions just adds :count n to the map maintained in "scheduled"

21. We'll use this function to implement incomplete-instruction?, incomplete-process?, and more-incomplete-processes? that we'll use later on:

```clojure
(defn incomplete-instruction? [instruction-w-count]
  (let [instr-effort (insts-effort (instruction-w-count :inst-type))
        instr-count (instruction-w-count :count)]
    (< instr-count instr-effort)))

(defn incomplete-process? [process-w-counts]
  (let [instrs-w-count (process-w-counts :instructions)]
    (some true? (map incomplete-instruction? instrs-w-count))))

(defn more-incomplete-processes? [processes-w-count]
  (some true? (map incomplete-process? processes-w-count)))
```

=> processes-w-count is just another name for the "scheduled" state map.
22. Diving deeper into the implementation, let’s now look at a single process and define a function that finds which instruction is to be fired if the scheduler decides to allocate a quantum to it. This translates to the first incomplete instruction if it is non-locked, that is, none of its locks have been set to locked by another process:

```clojure
(defn find-inst-to-be-fired-in-process
  [locks
   process-id
   the-process-instructions
   the-process-scheduled-parts]
  (let [p-not-locked-instrs (set (->> the-process-instructions
                                  (filter #(not (is-locked? process-id
                                                  the-process-instructions
                                                  locks
                                                  %)))))
        p-incomplete-instrs (set (->> (:instructions the-process-scheduled-parts)
                                 (filter incomplete-instruction?)
                                 (map #(dissoc % :count))))
        fireable-instrs (clojure.set/intersection p-not-locked-instrs
                                                 p-incomplete-instrs)
        instr-id-to-fire (->> fireable-instrs
                             (sort-by #(.indexOf the-process-instructions %) < )
                             (first)
                             (:inst-id))]
  instr-id-to-fire))
```

23. Now, let’s write `progress-on-process!`, which considers one particular process, fires its fireable instruction — as found by the preceding function, and updates all locks and scheduled states. This is quite a long function, as it is the heart of the scheduler:

```clojure
(defn progress-on-process!
  [locks-ref
   scheduled-ref
   the-process
   quantum]
  (let [the-process-instrs (the-process :instructions)
        processes-scheduled-parts (scheduled-processes-parts @ scheduled-ref)]
    )
```
Chapter 1

```clojure
(defn processes-scheduled-parts [processes]
  (let [process-id the-process
        process-instrs the-process-instrs
        process-scheduled-parts (->> processes
                                   (filter #(= (:process-id %) the-process))
                                   (first))]
    ;; Here we prepare the processes scheduled parts and take only
    ;; the relevant to the particular 'process-id'.
    (if-let [instr-to-fire-id (find-inst-to-be-fired-in-process @locks-ref
                                                              (:process-id the-process)
                                                              the-process-instrs
                                                              the-process-scheduled-parts)]
      ;; If there is one instruction in "process-id" to be fired;
      (dosync
       ;; We use the refs, because we need to do transactions involving
       ;; both "scheduled" and "locks"
       (let [instr-to-fire (->> the-process-instrs
                               (filter #(= (:inst-id %) instr-to-fire-id))
                               (first))]
         ;; We get the entry relevant to this instruction-id
         (cond
          (= (:inst-type instr-to-fire) :lock) (alter locks-ref lock
                                                  (:process-id the-process)
                                                  instr-to-fire-id)
          (= (:inst-type instr-to-fire) :unlock) (alter locks-ref unlock
                                                  (:process-id the-process)
                                                  instr-to-fire-id)
          (else nil)))))
    ;; If it is a "lock" or "unlock", We update the "locks" state
    ;; map
    (let [p-in-scheduled (->> @scheduled-ref
                            (filter #(= (:process-id %) the-process))
                            (first))]
      ;; To update the "scheduled" ref, we begin by finding the
      ;; ':process-d' in the processes vector
      (get p-in-scheduled :instructions)
      (filter #(= (:inst-id %) instr-to-fire-id)))))
```
Revisiting Arrays

;;;; Then We find the instruction in this process
idx-p-in-scheduled (max 0 (.indexOf @scheduled-ref p-in-scheduled))
idx-inst-in-p-in-scheduled (max 0 (.indexOf (get p-in-scheduled :instructions) instr-in-p-in-scheduled))
;; => We compute the index of the instruction; or we set it at 0
;;;; if it is not found, which means it is the first time it is
;;;; scheduled.
times-in-inst-in-p-in-scheduled (get
  (get (p-in-scheduled :instructions)
idx-inst-in-p-in-scheduled) :times )
;; => We get the times vector in "scheduled" related to this
;;;; instruction
    (conj times-in-inst-in-p-in-scheduled quantum)))
;; => And using assoc-in, with indices and keys as a "path
;;;; vector", we Update the "scheduled" ref with times vector
;;;; to which we Append the current "quantum".
true)
;; => If we were able to find a fireable instruction,
;;;; we issue "true".
false)))
;; => Else we issue "false".

24. The following functions will help us prepare empty locks and scheduled maps, which are to be used by progress-on-process!:
(defn prepare-scheduled
  [processes]
  (into []  (->> processes
    (map (fn [p] {:process-id (:process-id p)
                  :instructions (into []
                   (->> (:instructions p)
                    (map (fn [i] (assoc i
                      :times [])))))))))
    ;; => We prepare "scheduled" as being the same thing as the
    ;; "processes" map
    ;; with empty ":times" vectors added.)
25. Equipped with all these functions, we must address the problem of process selection for the allocation of each quantum of time. We must give each process an opportunity to access the scheduler quanta according to its priority. For this purpose, we will construct an infinite sequence of holding repetitions of a process ID as many times as their priority values. In this, a process with higher priority will always come before another with lower priority. Suppose we have process \( p_1 \) with priority 3, process \( p_2 \) with priority 2, and process \( p_3 \) with priority 1, then a sequence presenting the cycling that we described previously would be:

\[ p_1 \ p_1 \ p_1 \ p_2 \ p_2 \ p_3 \ p_1 \ p_1 \ p_1 \ p_2 \ p_2 \ p_3 \ldots \]

26. As the time quanums flow, the scheduler will have to pick at each step an element, cycling through the weighted cycling list, which we just saw, to be sure it is fair toward the process's priority.

27. The following functions create the priority-weighted cycling process IDs:

\begin{verbatim}
(defn gen-processes-cycles
 [processes]
  (let [sorted-procs-by-prio (sort-by :priority > processes)
     procs-pattern (mapcat #(repeat (:priority %)
                              %) sorted-procs-by-prio)]
    ;;=> A pattern is a single repetition "priority" times of each
    ;; process
    (cycle procs-pattern)))
  ;;=> Generates an infinite sequence like we described above.
\end{verbatim}
28. Locking programs may lead to infinite waiting. To tackle this problem, we will set a time-out for our scheduler, which will be twice the time needed by all the processes if they were to be executed sequentially, one after the other. This function does just that:

```clojure
(defn process-sequential-time
  [a-process]
  (let [instructions (a-process :instructions)
        inst-types (map :inst-type instructions)
        lengths (map #(get insts-effort %) inst-types)]
    (reduce + lengths)))
```

29. Finally, we can write our scheduler. While there are incomplete processes left to be scheduled and before the current quantum reaches time-out, the scheduler will cycle the weighted processes cycles, pick one process, and call `progress-on-a-process` on it. Note that we launch this on several programs as we are implementing time-sharing to do multithreading:

```clojure
(defn schedule-programs
  [language programs]
  (let [processes (into [] (map #(fire-a-process language
                                 (map :program %)
                                 (map :process-id %)
                                 (map :priority %))) programs))

  timeout (* 2 (reduce + (map process-sequential-time processes)))
  
  locks (ref (prepare-locks processes))
  scheduled (ref (prepare-scheduled processes))
  processes-cycles (gen-processes-cycles processes)]

  (loop [quantum 0
         remaining-processes processes-cycles]
    (if (more-incomplete-processes? (scheduled-processes-parts @scheduled))

    )
```
(< quantum timeout))

(do
  (progress-on-process! locks scheduled
    (first remaining-processes)
    quantum)
  ;; => progress on the selected process, with current "quantum"
  (recur (inc quantum)
    (next remaining-processes)))
  ;; => Go to next iteration, incrementing quantum and cycling
  ;; => through the The weighted processes cycles.
  @scheduled))

Now, let's define two random programs and see how they perform. First, define them in your REPL:

(def programs
  [({:priority 3,
      :program
      "heavy-op op1;light-op op2;lock l1;medium-op op3;unlock l1;",
      :process-id :pr1}
   {:priority 1,
      :program "lock l1;medium-op op4;unlock l1;medium-op op5;",
      :process-id :pr2}])

Now, launch schedule-programs:

(p (schedule-programs r3-language programs))

By launching it, you'll get the following output:

[{:process-id :pr1,
  :instructions
  [{:times [0 1 2 4 5 6 8 9 10 12],
    :inst-type :heavy-op,
    :inst-id "op1"}
   {:times [13 14], :inst-type :light-op, :inst-id "op2"}
   {:times [16], :inst-type :lock, :inst-id {:lock "l1"}}
   {:times [17 18 20 21 22], :inst-type :medium-op, :inst-id "op3"}
   {:times [24], :inst-type :unlock, :inst-id {:unlock "l1"}}]
   {:process-id :pr2,
    :instructions
    [{:times [3], :inst-type :lock, :inst-id {:lock "l1"}}]
Revisiting Arrays

{:times [7 11 15 27 31], :inst-type :medium-op, :inst-id "op4"}
{:times [35], :inst-type :unlock, :inst-id {:unlock "l1"}}
{:times [39 43 47 51 55], :inst-type :medium-op, :inst-id "op5"}]

Simulating a call stack using arrays

A call stack is a data structure that is built when a program runs. As function calls keep coming in, the information regarding their code is arranged in frames, that is, a frame per call or variable evaluation. And these frames are stacked up. The program execution is then a matter of "unwinding" these frames, that is, after a frame at the top of the stack has been evaluated, it is unstacked and the process resumes at the new frame that is now at the top of the call stack.

Here we will observe a simple rule to unwind: as the execution goes, if we unstack a variable, we store it, and when we encounter a function call to unstack, we store the return value of its call and pass to it the parameters that we’ve stored so far. The next figure explains this process:

Unwinding the frames in a call Stack
How to do it...

1. First of all, let’s define our namespace incorporating all Clojure facilities that we will use:
   
   ```clojure
   (ns recipe4.core
     (:require [instaparse.core :as insta])
   ;;=> To parse our programs
     (:require [clojure.zip :as z])
   ;;=> To walk and process parse trees
     (:require [clojure.pprint :refer :all])
   ;;=> To pretty print results
     (:require [clojure.walk :as walk])
   ;;=> To transform some nodes
     ;; in our programs' parse trees
   ```

2. We’ll also alias clojure.pprint/pprint so that we can easily pretty-print the results of our computations:
   
   ```clojure
   (def p pprint)
   ```

3. We’ll design a minimal language that will be parsed with instaparse.

   Instaparse (https://github.com/engelberg/instaparse) is a parser generator written in Clojure. Explaining the mechanism of Instaparse is beyond the scope of this book, but you should know it handles context-free grammars (CFG), and generates parse trees of your input programs according to these grammar concepts.

4. Our language will only be able to understand function calls. You can think of it as a kind of Lisp, but with no prefix notation; you can write your functions using the old mathematical way in this. Besides, our language is able to understand function declarations. Here is an example of what a program in this language looks like:
   
   ```clojure
   decl-fn f(x,y){
     plus(x,y);
   };
   plus(f(1,2),f(3,4));
   ```

5. The functions without declarations are considered primitive or library functions in our programs.
6. Here is the instaparse grammar that is able to parse programs written in our minimal language:

```
(def r4-language
  "S = ((FN-CALL|FN-DECL) <FN-SEP>)*
  FN-CALL = <optional-whitespace> ID <optional-whitespace> <left-paren> PARAMS-LIST <right-paren>
  PARAMS-LIST = <optional-whitespace> (ID|FN-CALL)
  <optional-whitespace> (ID|FN-CALL))
  FN-DECL = <optional-whitespace> 'decl-fn'
  <whitespace> ID <optional-whitespace> <left-paren> ARGS-LIST <right-paren> <optional-whitespace>
  <left-curly> FN-DECL-BODY <right-curly>
  ARGS-LIST = ID <optional-whitespace> (ID|FN-CALL) <optional-whitespace> ID
  <optional-whitespace> (ID|FN-CALL) <optional-whitespace> ID
  FN-DECL-BODY = (FN-CALL <FN-SEP>)*
  left-paren = '('
  right-paren = ')'
  left-curly = '{'
  right-curly = '}'
  ID = #'[a-zA-Z0-9]+'
  whitespace = '#\s+'
  optional-whitespace = '#\s*' 
  FN-SEP = <optional-whitespace> ';' <optional-whitespace>
  PARAMS-SEP = <optional-whitespace> ',' <optional-whitespace>
")
```

7. Note that identifiers between angle brackets will not be shown in the parse tree, so there's no use of referring to white-space tags, for instance.

8. Let's see what the parse tree of the program we previously wrote looks like. Issue the following code in your REPL:

```
(p (insta/parse (insta/parser r4-language) "decl-fn f(x,y){
    plus(x,y);
  };
  plus(f(1,2),f(3,4));")
```

9. After this step, you'll get the following output:

```
[:S
 [:FN-DECL
  "decl-fn"
  [:ID "f"]
  [:ARGS-LIST [:ID "x"] [:ID "y"]]
  [:FN-DECL-BODY
   [:FN-CALL [:ID "plus"] [:PARAMS-LIST [:ID "x"] [:ID "y"]]]]]
 [:FN-CALL
  [:ID "plus"]
  [:PARAMS-LIST
   [:FN-CALL [:ID "f"] [:PARAMS-LIST [:ID "1"] [:ID "2"]]]
   [:FN-CALL [:ID "f"] [:PARAMS-LIST [:ID "3"] [:ID "4"]]]]]
```

10. Now we'll use the instaparse and transform functions to provide a more convenient representation of our parsed program. The transform function replaces particular tags in the parse tree, applying a function to the rest of elements in the vector that contains those tags. Here is how we want to transform the parse trees:

```
(defn gen-program
 [parser program]
 (into [] (insta/transform
 { :S (fn [ & args] args) :FN-CALL (fn [fn-id params] [:FN-CALL (fn-id 1) params])
  :PARAMS-LIST (fn[& params] (into [] params))
  :FN-DECL (fn [_ decl-fn-id args body] [:FN-DECL (decl-fn-id 1) args body])
  :ARGS-LIST (fn [& args] (into [] args))
  :FN-DECL-BODY (fn [& body] (into [] body))} (parser program))))
```

11. To better understand what this function does you can refer to its output, which is as follows. Input the following code into your REPL:

```
(p (gen-program (insta/parser r4-language) "decl-fn f(x,y){
  plus(x,y);
};
plus(f(1,2),f(3,4));") )
```
12. After completing this step, you'll get the following output:

```
[[:FN-DECL
  "f"
  [[:ID "x"] [:ID "y"]]
  [[:FN-CALL "plus" [[:ID "x"] [:ID "y"]]]]]
[[:FN-CALL
  "plus"
  [[:FN-CALL "f" [[:ID "1"] [:ID "2"]]]
  [[:FN-CALL "f" [[:ID "3"] [:ID "4"]]]]]
```

13. With this representation of our program, we first need to know which functions are declared:

```
(defn get-fn-decls
  [program]
  (->> program
    (filter #(= :FN-DECL (get % 0)))
    ;;=> Take only instructions with :FN-DECL tag
    (into []))
```

14. Complementary to this function, we need a function that tells us which instructions (function calls) we have in our program:

```
(defn get-instructions
  [program]
  (->> program
    (filter #(not= :FN-DECL (get % 0)))
    ;;=> Take only instructions with no :FN-DECL tag.
    (into []))
```

15. Now we will focus on how to translate declared function calls. We need to exchange the reference to such calls with the bodies of declaration, in which we inject the parameters passed along with the call. Let's first see the declaration of a particular function:

```
(defn get-fn-id-decl
  [fn-decls fn-id]
  (->> fn-decls
    (filter # (= (get % 1) fn-id))
    ;;=> Returns the fn-decl that matches the passed fn-id.
    (first))
  ;;=> This function will return 'nil' if there is no
  ;; declaration found for it.
16. Now we are going to implement call-fn, which is a function that does the actual translation of a function call using its declaration (if we ever find any) and passed parameters:

```(defn call-fn
 [fn-decl fn-call]
 (let [decl-args-list (fn-decl 2)
 ;;=> we get the args in the declaration
   decl-body (fn-decl 3)
 ;;=> We get the body of the declaration.
   fn-call-params (fn-call 2)]
 ;;=> We get the passed parameters
   (if (not (= (count decl-args-list) (count fn-call-params)))
   [:arity-error-in-calling (fn-decl 1 )])
 ;;=> If the count of parameters and args mismatch, we say we have an arity error
   (let [replacement-map (zipmap decl-args-list fn-call-params)]
 ;;=> we prepare a replacement map for 'postwalk-replace':
 ;;  zipmap builds a map containing keys from the first seq
 ;;  'decl-args-list' and vals from the second one 'fn-call-params'.
     (walk/postwalk-replace replacement-map decl-body)))))
;;=> 'postwalk-replace' will then change in 'decl-body' the
;; arguments 'decl-args-list' by corresponding parameters in
;; 'fn-call-params'
```

17. Next, we will do the actual translation of the declared function calls and leave the non-declared functions as they are, assuming that they are primitive or library functions. This is why we called the expand-to-primitive-calls function:

```(defn expand-to-primitive-calls
 [program] ;;=> A program generated with 'gen-program'
 (let [fn-decls (get-fn-decls program)
   instructions (get-instructions program)
 ;;=> preparing function declarations and instructions .
   zpr (z/vector-zip instructions)]
 ;;=> A zipper to walk instructions.
 (loop [result instructions

 ;;=> We initially have our result set to be our instructions.
   loc (-> zpr z/down)]
 (if (-> loc z/end?)
   result
 ;;=> end of recursion. If no more nodes to visit, we emit result.
   (let [current-node (-> loc z/node)]
 ;;=> We store current node
 ```
Revisiting Arrays

(if (= (get current-node 0 :FN-CALL))
  ;;=> If it is a function call
  (if-let [the-decl (get-fn-id-decl fn-decls (get current-node 1))]
    ;;=> and it has a declaration associated with it
    (recur (walk/postwalk-replace {(-> loc z/node)
      (call-fn the-decl current-node)}
      result )
    (-> loc z/next))
  ;;=> we recur replacing this current-node with
  ;; the function declaration along with the parameters.
    (recur result (-> loc z/next))
  ;;=> else we recur leaving the function as is considering it
  ;; to be 'primitive'.
    (recur result (-> loc z/next)))))
  ;;=> or we recur leaving the instruction as is, because here
  ;; we only have a variable evaluation.

18. At this particular point we are able to construct a call stack for an instruction:

(defn a-call-stack
  [a-call]
  (let [zpr (z/vector-zip a-call)]
    ;;=> A zipper to walk our call.
    (loop [result []
      loc (-> zpr z/down)]
      ;;=> End of the recursion, we emit result.
      (let [current-node (-> loc z/node)]
        result)
      ;;=> we store the current node.
        (recur (if (and
          (not (vector? current-node))
          (not= :FN-CALL current-node)
          (not= :ID current-node))
          ;;=> If this is a literal, that is, not a vector, and not a tag,
          (conj result {(-> loc z/left z/node) current-node})
          ;;=> I add it to the stack, along with the node at its left;
          ;;=> for instance, we'll have {:ID a-value}
          ;; or {:FN-CALL a value}
          result)
          ;; => Else we leave the stack as is.
          (-> loc z/next)))))
    ; and we go to the next node.
19. Finally, we will get to construct a stack for every instruction:

\[
(defn program-call-stack
  [prog]
  (into []
    (map a-call-stack
      (expand-to-primitive-calls prog))))
\]

Let's see how it works. Type the following in to your REPL:

\[
(p  (program-call-stack (gen-program (insta/parser r4-language)
  "decl-fn f(x,y) {
    plus(x,y);
  };
  plus(f(1,2),f(3,4));
  f(4,5);"  )))
\]

The result of this would be:

\[
[[{:FN-CALL "plus"} 
  {:FN-CALL "plus"} 
  {:ID "1"} 
  {:ID "2"} 
  {:FN-CALL "plus"} 
  {:ID "3"} 
  {:ID "4"}]] 

[[{:FN-CALL "plus"} {:ID "4"} {:ID "5"}] ]
\]

Here, the stack top comes last, as vectors in Clojure are way more efficiently accessed from the tail. This stack would be unwinded as follows:

1. This stack processes instruction 1.
2. Then it stores the value 4.
3. Stores the values 3,4.
4. Stores the value of plus("3","4").
5. Stores the values of 2,plus("3","4").
6. Stores the values of 1,2,plus("3","4").
7. Stores the values of plus("1","2"),plus("3","4").
8. Stores the values of plus(plus("1","2"),plus("3","4")).
Revisiting Arrays

9. Instruction 1 finishes returning \( \text{plus}(\text{plus}("1","2"), \text{plus}("3","4")) \).
10. Then it processes instruction 2.
11. Stores the value 5.
12. Stores the values 4,5.
13. Stores the value of \( \text{plus} ("4","5") \).
14. Instruction 2 finishes returning the value of \( \text{plus} ("4","5") \).
Where to buy this book

You can buy Clojure Data Structures and Algorithms Cookbook 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.