Extracted from:

Programming Elixir

Functional |> Concurrent |> Pragmatic |> Fun

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Programming Elixir

Functional
|> Concurrent
|> Pragmatic
|> Fun

Dave Thomas

Foreword by
José Valim,
Creator of Elixir

edited by Lynn Beighley

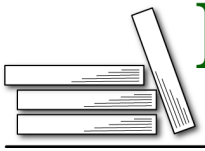
Programming Elixir

Functional |> Concurrent |> Pragmatic |> Fun

Dave Thomas

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Lynn Beighley (editor)
Potomac Indexing, LLC (indexer)
Candace Cunningham (copyeditor)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-937785-58-1
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—October, 2014

In this chapter, you'll see:

- The recursive structure of lists
- Traversing and building lists
- Accumulators
- Implementing map and reduce

CHAPTER 7

Lists and Recursion

When we program with lists in conventional languages, we treat them as things to be iterated—it seems natural to loop over them. So why do we have a chapter on *lists and recursion*? Because if you look at the problem in the right way, recursion is a perfect tool for processing lists.

Heads and Tails

Earlier we said a list may either be empty or consist of a head and a tail. The head contains a value and the tail is itself a list. This is a recursive definition.

We'll represent the empty list like this: `[ ]`.

Let's imagine we could represent the split between the head and the tail using a pipe character, `|`. The single element list we normally write as `[3]` can be written as the value `3` joined to the empty list:

```
[ 3  |  [] ]
```

When we see the pipe character, we say that what is on the left is the head of a list and what's on the right is the tail.

Let's look at the list `[2, 3]`. The head is `2`, and the tail is the single-element list containing `3`. And we know what that list looks like—it is our previous example. So we could write `[2,3]` as

```
[ 2  |  [ 3 | [] ] ]
```

At this point, part of your brain is telling you to go read today's XKCD—this list stuff can't be useful. Ignore that small voice, just for a second. We're about to do something magical. But before we do, let's add one more term, making our list `[1, 2, 3]`. This is the head `1` followed by the list `[2, 3]`, which is what we derived a moment ago:

```
[ 1  |  [ 2  |  [ 3 | [] ] ] ]
```

This is valid Elixir syntax. Type it into iex.

```
iex> [ 1  |  [ 2  |  [ 3 | [] ] ] ]
[1, 2, 3]
```

And here's the magic. When we discussed pattern matching, we said the pattern could be a list, and the values in that list would be assigned from the right-hand side.

```
iex> [a, b, c ] = [ 1, 2, 3 ]
[1, 2, 3]
iex> a
1
iex> b
2
iex> c
3
```

We can also use the pipe character in the pattern. What's to the left of it matches the head value of the list, and what's to the right matches the tail.

```
iex> [ head | tail ] = [ 1, 2, 3 ]
[1, 2, 3]
iex> head
1
iex> tail
[2, 3]
```

Using Head and Tail to Process a List

Now we can split a list into its head and its tail, and we can construct a list from a value and a list, which become the head and tail of that new list.

So why talk about lists after we talk about modules and functions? Because lists and recursive functions go together like fish and chips. Let's look at finding the length of a list.

- The length of an empty list is 0.
- The length of a list is 1 plus the length of that list's tail.

Writing that in Elixir is easy:

lists/mylist.exs

```
defmodule MyList do
  def len([]), do: 0
  def len([head|tail]), do: 1 + len(tail)
end
```

The only tricky part is the definition of the function's second variant:

```
def len([ head | tail ]) ...
```

How iex Displays Lists

In Chapter 11, *Strings and Binaries*, on page ?, you'll see that Elixir has two representations for strings. One is the familiar sequence of characters in consecutive memory locations. Literals written with double quotes use this form.

The second form, using single quotes, represents strings as a list of integer codepoints. So the string 'cat' is the three codepoints: 99, 97, and 116.

This is a headache for iex. When it sees a list like [99,97,116] it doesn't know if it is supposed to be the string 'cat' or a list of three numbers. So it uses a heuristic. If all the values in a list represent printable characters, it displays the list as a string; otherwise it displays a list of integers.

```
iex> [99, 97, 116]
'cat'
iex> [99, 97, 116, 0]   # '0' is nonprintable
[99, 97, 116, 0]
```

In Chapter 11, *Strings and Binaries*, on page ?, we'll cover how to bypass this behavior. In the meantime, don't be surprised if a string pops up when you were expecting a list.

This is a pattern match for any nonempty list. When it does match, the variable head will hold the value of the first element of the list, and tail will hold the rest of the list. (And remember that every list is terminated by an empty list, so the tail can be [].)

Let's see this at work with the list [11, 12, 13, 14, 15]. At each step, we take off the head and add 1 to the length of the tail:

```
len([11,12,13,14,15])
= 1 + len([12,13,14,15])
= 1 + 1 + len([13,14,15])
= 1 + 1 + 1 + len([14,15])
= 1 + 1 + 1 + 1 + len([15])
= 1 + 1 + 1 + 1 + 1 + len([])
= 1 + 1 + 1 + 1 + 1 + 0
= 5
```

Let's try our code to see if theory works in practice:

```
iex> c "mylist.exs"
...mylist.exs:3: variable head is unused
[MyList]
iex> MyList.len([])
0
iex> MyList.len([11,12,13,14,15])
5
```

It works, but we have a compilation warning—we never used the variable head in the body of our function. We can fix that, and make our code more explicit, using the special variable _ (underscore), which acts as a placeholder. We can also use an underscore in front of any variable name to turn off the warning if that variable isn't used. I sometimes like to do this to document the unused parameter.

lists/mylist1.exs
```
defmodule MyList do
  def len([]),             do: 0
  def len([_head | tail]), do: 1 + len(tail)
end
```

When we compile, the warning is gone. (However, if you compile the second version of MyList, you may get a warning about "redefining module MyList." This is just Elixir being cautious.)

```
iex> c "mylist1.exs"
[MyList]
iex> MyList.len([1,2,3,4,5])
5
iex> MyList.len(["cat", "dog"])
2
```

Using Head and Tail to Build a List

Let's get more ambitious. Let's write a function that takes a list of numbers and returns a new list containing the square of each. We don't show it, but these definitions are also inside the MyList module.

lists/mylist1.exs
```
def square([]),              do: []
def square([ head | tail ]), do: [ head*head | square(tail) ]
```

There's a lot going on here. First, look at the parameter patterns for the two definitions of square. The first matches an empty list and the second matches all other lists.

Second, look at the body of the second definition:

```
def square([ head | tail ]), do: [ head*head | square(tail) ]
```

When we match a nonempty list, we return a new list whose head is the square of the original list's head and whose tail is list of squares of the tail. This is the recursive step.

Let's try it:

```
iex> c "mylist1.exs"
[MyList]
iex> MyList.square []        # this calls the 1st definition
[]
iex> MyList.square [4,5,6]   # and this calls the 2nd
[16, 25, 36]
```

Let's do something similar—a function that adds 1 to every element in the list:

lists/mylist1.exs

```
def add_1([]),              do: []
def add_1([ head | tail ]), do: [ head+1 | add_1(tail) ]
```

And call it:

```
iex> c "mylist1.exs"
[MyList]
iex> MyList.add_1 [1000]
[1001]
iex> MyList.add_1 [4,6,8]
[5, 7, 9]
```

Creating a Map Function

With both square and add_1, all the work is done in the second function definition. And that definition looks about the same for each—it returns a new list whose head is the result of either squaring or incrementing the head of its argument and whose tail is the result of calling itself recursively on the tail of the argument. Let's generalize this. We'll define a function called map that takes a list and a function and returns a new list containing the result of applying that function to each element in the original.

lists/mylist1.exs

```
def map([], _func),             do: []
def map([ head | tail ], func), do: [ func.(head) | map(tail, func) ]
```

The map function is pretty much identical to the square and add_1 functions. It returns an empty list if passed an empty list; otherwise it returns a list where the head is the result of calling the passed-in function and the tail is a recursive call to itself. Note that in the case of an empty list, we use _func as the second parameter. The underscore prevents Elixir from warning us about an unused variable.

To call this function, pass in a list and a function (defined using fn).

```
iex> c "mylist1.exs"
[MyList]
iex> MyList.map [1,2,3,4], fn (n) -> n*n end
[1, 4, 9, 16]
```

A function is just a built-in type, defined between fn and the end. Here we pass a function as the second argument (func) to map. This is invoked inside map using func.(head), which squares the value in head, using the result to build the new list.

We can call map with a different function:

```
iex> MyList.map [1,2,3,4], fn (n) -> n+1 end
[2, 3, 4, 5]
```

And another:

```
iex> MyList.map [1,2,3,4], fn (n) -> n > 2 end
[false, false, true, true]
```

And we can do the same using the & shortcut notation:

```
iex> MyList.map [1,2,3,4], &(&1 + 1)
[2, 3, 4, 5]
iex> MyList.map [1,2,3,4], &(&1 > 2)
[false, false, true, true]
```

Keeping Track of Values During Recursion

So far you've seen how to process each element in a list, but what if we want to sum all of the elements? The difference here is that we need to remember the partial sum as we process each element in turn.

In terms of a recursive structure, it's easy:

- sum([]) → 0
- sum([head |tail]) → «*total*» + sum(tail)

But the basic scheme gives us nowhere to record the total as we go along. Remember that one of our goals is to have immutable state, so we can't keep the value in a global or module-local variable.

But we *can* pass the state in a function's parameter.

lists/sum.exs

```
defmodule MyList do
  def sum([], total),              do: total
  def sum([ head | tail ], total), do: sum(tail, head+total)
end
```

Our sum function now has two parameters, the list and the total so far. In the recursive call, we pass it the list's tail and increment the total by the value of the head.

At all times, these types of functions maintain an *invariant*, a condition that is true on return from any call (or nested call). In this case, the invariant is that at any stage of the recursion, the sum of the elements in the list parameter plus the current total will equal the total of the entire list. Thus, when the list becomes empty the total will be the value we want.

When we call sum we have to remember to pass both the list and the initial total value (which will be 0):

```
iex> c "sum.exs"
[MyList]
iex> MyList.sum([1,2,3,4,5], 0)
15
iex> MyList.sum([11,12,13,14,15], 0)
65
```

Having to remember that extra zero is a little tacky, so the convention in Elixir is to hide it—our module has a public function that takes just a list, and it calls private functions to do the work.

lists/sum2.exs

```
defmodule MyList do

  def sum(list), do: _sum(list, 0)

  # private methods
  defp _sum([], total),              do: total
  defp _sum([ head | tail ], total), do: _sum(tail, head+total)
end
```

Two things to notice here: First, we use defp to define a private function. You won't be able to call these functions outside the module. Second, we chose to give our helper functions the same name as our public function, but with a leading underscore. Elixir treats them as being independent, but a human reader can see that they are clearly related.

(Had we kept the *exact* same name, they would still be different functions. as they have a different arity from the original sum function. The leading underscore simply makes it explicit. Some library code also uses do_*xxx* for these helpers.)

Your Turn

➤ *Exercise: ListsAndRecursion-0*

I defined our sum function to carry a partial total as a second parameter so I could illustrate how to use accumulators to build values. The sum function can also be written without an accumulator. Can you do it?

Generalizing Our Sum Function

The sum function reduces a collection to a single value. Clearly other functions need to do something similar—return the greatest/least value, the product of the elements, a string containing the elements with spaces between them, and so on. How can we write a general-purpose function that reduces a collection to a value?

We know it has to take a collection. We also know we need to pass in some initial value (just like our sum/1 function passed a 0 as an initial value to its helper). Additionally, we need to pass in a function that takes the current value of the reduction along with the next element of the collection, and returns the next value of the reduction. So, it looks like our reduce function will be called with three arguments:

```
reduce(collection, initial_value, fun)
```

Now let's think about the recursive design:

- reduce([], value, _) → value
- reduce([head |tail], value, fun) → reduce(tail, fun.(head, value), fun)

reduce applies the function to the list's head and the current value, and passes the result as the new current value when reducing the list's tail.

Here's our code for reduce. See how closely it follows the design.

lists/reduce.exs

```
defmodule MyList do
  def reduce([], value, _) do
    value
  end
  def reduce([head | tail], value, func) do
    reduce(tail, func.(head, value), func)
  end
end
```

And, again, we can use the shorthand notation to pass in the function:

```
iex> c "reduce.exs"
[MyList]
```

```
iex> MyList.reduce([1,2,3,4,5], 0, &(&1 + &2))
15
iex> MyList.reduce([1,2,3,4,5], 1, &(&1 * &2))
120
```

Your Turn

➤ *Exercise: ListsAndRecursion-1*

Write a mapsum function that takes a list and a function. It applies the function to each element of the list and then sums the result, so

```
iex> MyList.mapsum [1, 2, 3], &(&1 * &1)
14
```

➤ *Exercise: ListsAndRecursion-2*

Write a max(list) that returns the element with the maximum value in the list. (This is slightly trickier than it sounds.)

➤ *Exercise: ListsAndRecursion-3*

An Elixir single-quoted string is actually a list of individual character codes. Write a caesar(list, n) function that adds n to each list element, wrapping if the addition results in a character greater than *z*.

```
iex> MyList.caesar('ryvkve', 13)
?????? :)
```

More Complex List Patterns

Not every list problem can be easily solved by processing one element at a time. Fortunately, the join operator, |, supports multiple values to its left. Thus, you could write

```
iex> [ 1, 2, 3 | [ 4, 5, 6 ]]
[1, 2, 3, 4, 5, 6]
```

The same thing works in patterns, so you can match multiple individual elements as the head. For example, the following program swaps pairs of values in a list.

lists/swap.exs

```
defmodule Swapper do
  def swap([]), do: []
  def swap([ a, b | tail ]), do: [ b, a | swap(tail) ]
  def swap([_]), do: raise "Can't swap a list with an odd number of elements"
end
```

We can play with it in iex:

```
iex> c "swap.exs"
[Swapper]
iex> Swapper.swap [1,2,3,4,5,6]
[2, 1, 4, 3, 6, 5]
iex> Swapper.swap [1,2,3,4,5,6,7]
** (RuntimeError) Can't swap a list with an odd number of elements
```

The third definition of `swap` matches a list with a single element. This will happen if we get to the end of the recursion and have only one element left. As we take two values off the list on each cycle, the initial list must have had an odd number of elements.

Lists of Lists

Let's imagine we had recorded temperatures and rainfall at a number of weather stations. Each reading looks like this:

```
[ timestamp, location_id, temperature, rainfall ]
```

Our code is passed a list containing a number of these readings, and we want to report on the conditions for one particular location, number 27.

lists/weather.exs

```
defmodule WeatherHistory do

  def for_location_27([]), do: []
  def for_location_27([ [time, 27, temp, rain ] | tail]) do
    [ [time, 27, temp, rain] | for_location_27(tail) ]
  end
  def for_location_27([ _ | tail]), do: for_location_27(tail)

end
```

This is a standard *recurse until the list is empty* stanza. But look at our function definition's second clause. Where we'd normally match into a variable called `head`, here the pattern is

```
for_location_27([ [ time, 27, temp, rain ] | tail])
```

For this to match, the head of the list must itself be a four-element list, and the second element of this sublist must be 27. This function will execute only for entries from the desired location. But when we do this kind of filtering, we also have to remember to deal with the case when our function doesn't match. That's what the third line does. We could have written

```
for_location_27([ [ time, _, temp, rain ] | tail])
```

but in reality we don't care *what* is in the head at this point.

In the same module we define some simple test data:

lists/weather.exs
```
def test_data do
  [
   [1366225622, 26, 15, 0.125],
   [1366225622, 27, 15, 0.45],
   [1366225622, 28, 21, 0.25],
   [1366229222, 26, 19, 0.081],
   [1366229222, 27, 17, 0.468],
   [1366229222, 28, 15, 0.60],
   [1366232822, 26, 22, 0.095],
   [1366232822, 27, 21, 0.05],
   [1366232822, 28, 24, 0.03],
   [1366236422, 26, 17, 0.025]
  ]
end
```

We can use that to play with our function in iex. To make this easier, I'm using the import function. This adds the functions in WeatherHistory to our local name scope. After calling import we don't have to put the module name in front of every function call.

```
iex> c "weather.exs"
[WeatherHistory]
iex> import WeatherHistory
nil
iex> for_location_27(test_data)
[[1366225622, 27, 15, 0.45], [1366229222, 27, 17, 0.468],
 [1366232822, 27, 21, 0.05]]
```

Our function is specific to a particular location, which is pretty limiting. We'd like to be able to pass in the location as a parameter. We can use pattern matching for this.

lists/weather2.exs
```
defmodule WeatherHistory do

  def for_location([], _target_loc), do: []

➤ def for_location([ [time, target_loc, temp, rain ] | tail], target_loc) do
    [ [time, target_loc, temp, rain] | for_location(tail, target_loc) ]
  end

  def for_location([ _ | tail], target_loc), do: for_location(tail, target_loc)

end
```

Now the second function fires only when the location extracted from the list head equals the target location passed as a parameter.

But we can improve on this. Our filter doesn't care about the other three fields in the head—it just needs the location. But we do need the value of the head itself to create the output list. Fortunately, Elixir pattern matching is recursive and we can match patterns inside patterns.

lists/weather3.exs

```
defmodule WeatherHistory do

  def for_location([], target_loc), do: []

➤ def for_location([ head = [_, target_loc, _, _ ] | tail], target_loc) do
    [ head | for_location(tail, target_loc) ]
  end

  def for_location([ _ | tail], target_loc), do: for_location(tail, target_loc)

end
```

The key change here is this line:

```
def for_location([ head = [_, target_loc, _, _ ] | tail], target_loc)
```

Compare that with the previous version:

```
def for_location([ [ time, target_loc, temp, rain ] | tail], target_loc)
```

In the new version, we use placeholders for the fields we don't care about. But we also match the entire four-element array into the parameter `head`. It's as if we said "match the head of the list where the second element is matched to `target_loc` and then match that whole head with the variable `head`." We've extracted an individual component of the sublist as well as the entire sublist.

In the original body of `for_location`, we generated our result list using the individual fields:

```
def for_location([ [ time, target_loc, temp, rain ] | tail], target_loc)
  [ [ time, target_loc, temp, rain ] | for_location(tail, target_loc) ]
end
```

In the new version, we can just use the `head`, making it a lot more clear:

```
def for_location([ head = [_, target_loc, _, _ ] | tail], target_loc)  do
  [ head | for_location(tail, target_loc) ]
end
```

Your Turn

➤ *Exercise: ListsAndRecursion-4*

Write a function `MyList.span(from, to)` that returns a list of the numbers from `from` up to `to`.

The List Module in Action

The List module provides a set of functions that operate on lists.

```
#
# Concatenate lists
#
iex> [1,2,3] ++ [4,5,6]
[1, 2, 3, 4, 5, 6]
#
# Flatten
#
iex> List.flatten([[[1], 2], [[[3]]]])
[1, 2, 3]
#
# Folding (like reduce, but can choose direction)
#
iex> List.foldl([1,2,3], "", fn value, acc -> "#{value}(#{acc})" end)
"3(2(1()))"
iex> List.foldr([1,2,3], "", fn value, acc -> "#{value}(#{acc})" end)
"1(2(3()))"
#
# Merging lists and splitting them apart
#
iex> l = List.zip([[1,2,3], [:a,:b,:c], ["cat", "dog"]])
[{1, :a, "cat"}, {2, :b, "dog"}]
iex> List.unzip(l)
[[1, 2], [:a, :b], ["cat", "dog"]]
#
# Accessing tuples within lists
#
iex> kw = [{:name, "Dave"}, {:likes, "Programming"}, {:where, "Dallas", "TX"}]
[{:name, "Dave"}, {:likes, "Programming"}, {:where, "Dallas", "TX"}]
iex> List.keyfind(kw, "Dallas", 1)
{:where, "Dallas", "TX"}
iex> List.keyfind(kw, "TX", 2)
{:where, "Dallas", "TX"}
iex> List.keyfind(kw, "TX", 1)
nil
iex> List.keyfind(kw, "TX", 1, "No city called TX")
"No city called TX"
iex> kw = List.keydelete(kw, "TX", 2)
[name: "Dave", likes: "Programming"]
iex> kw = List.keyreplace(kw, :name, 0, {:first_name, "Dave"})
[first_name: "Dave", likes: "Programming"]
```

Get Friendly with Lists

Lists are the natural data structure to use when you have a stream of values to handle. You'll use them to parse data, handle collections of values, and record the results of a series of function calls. It's worth spending a while getting comfortable with them.

Next we'll look at the various dictionary types, including maps. These let us organize data into collections of key/value pairs.