Functional thinking: Functional features in Groovy, Part 1

Treasures lurking in Groovy

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

22 November 2011

Over time, languages and runtimes have handled more and more mundane details for us. Functional languages exemplify this trend, but modern dynamic languages have also incorporated many functional features to make developers' lives easier. This installment investigates some of the functional features already lurking in Groovy, showing how recursion hides state and how to build lazy lists.

View more content in this series

About this series

This series aims to reorient your perspective toward a functional mindset, helping you look at common problems in new ways and find ways to improve your day-to-day coding. It explores functional programming concepts, frameworks that allow functional programming within the Java™ language, functional programming languages that run on the JVM, and some future-leaning directions of language design. The series is geared toward developers who know Java and how its abstractions work but have little or no experience using a functional language.

Confession: I never want to work in a non-garbage-collected language again. I paid my dues in languages like C++ for too many years, and I don't want to surrender the conveniences of modern languages. That's the story of how software development progresses. We build layers of abstraction to handle (and hide) mundane details. As the capabilities of computers have grown, we've offloaded more tasks to languages and runtimes. As recently as a decade ago, developers shunned interpreted languages for being too slow for production applications, but they are common now. Many of the features of functional languages were prohibitively slow a decade ago but make perfect sense now because they optimize developer time and effort.

Many of the features I cover in this article series show how functional languages and frameworks handle mundane details. However, you don't have to go to a functional language to start reaping benefits from functional constructs. In this installment and the next, I'll show how some functional programming has already crept into Groovy.

Groovy's functional-ish lists

Groovy significantly augments the Java collection libraries, including adding functional constructs. The first favor Groovy does for you is provide a different perspective on lists, which seems trivial at first but offers some interesting benefits.

Seeing lists differently

If your background is primarily in C or C-like languages (including Java), you probably conceptualize lists as indexed collections. This perspective makes it easy to iterate over a collection, even when you don't explicitly use the index, as shown in the Groovy code in Listing 1:

Listing 1. List traversal using (hidden) indexes

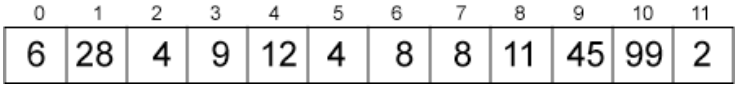
```
def perfectNumbers = [6, 28, 496, 8128]

def iterateList(listOfNums) {
  listOfNums.each { n ->
    println "${n}"
  }
}

iterateList(perfectNumbers)
```

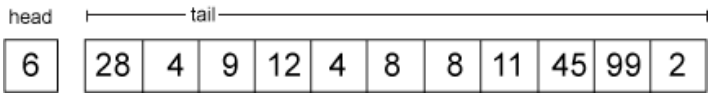
Groovy also includes an `eachWithIndex()` iterator, which provides the index as a parameter to the code block for cases in which explicit access is necessary. Even though I don't use an index in the `iteraterList()` method in Listing 1, I still think of it as an ordered collection of slots, as shown in Figure 1:

Figure 1. Lists as indexed slots



Many functional languages have a slightly different perspective on lists, and fortunately Groovy shares this perspective. Instead of thinking of a list as indexed slots, think of it as a combination of the first element in the list (the *head*) plus the remainder of the list (the *tail*), as shown in Figure 2:

Figure 2. A list as its head and tail



Thinking about a list as head and tail allows me to iterate through it using recursion, as shown in Listing 2:

Listing 2. List traversal using recursion

```
def recurseList(listOfNums) {
  if (listOfNums.size == 0) return;
    println "${listOfNums.head()}"
    recurseList(listOfNums.tail())
}

recurseList(perfectNumbers)
```

In the `recurseList()` method in Listing 2, I first check to see if the list that's passed as the parameter has no elements in it. If that's the case, then I'm done and can return. If not, I print out the first element in the list, available via Groovy's `head()` method, and then recursively call the `recurseList()` method on the remainder of the list.

Recursion has technical limits built into the platform (see Resources), so this isn't a panacea. But it should be safe for lists that contain a small number of items. I'm more interested in investigating the impact on the structure of the code, in anticipation of the day when the limits ease or disappear. Given the shortcomings, the benefit of the recursive version may not be immediately obvious. To make it more so, consider the problem of filtering a list. In Listing 3, I show an example of a filtering method that accepts a list and a predicate (a boolean test) to determine if the item belongs in the list:

Listing 3. Imperative filtering with Groovy

```
def filter(list, p) {
  def new_list = []
  list.each { i ->
    if (p(i))
      new_list << i
  }
  new_list
}

modBy2 = { n -> n % 2 == 0}

l = filter(1..20, modBy2)
```

The code in Listing 3 is straightforward: I create a holder variable for the elements that I want to keep, iterate over the list, check each element with the inclusion predicate, and return the list of filtered items. When I call `filter()`, I supply a code block specifying the filtering criteria.

Consider a recursive implementation of the filter method from Listing 3, shown in Listing 4:

Listing 4. Recursive filtering with Groovy

```
def filter(list, p) {
  if (list.size() == 0) return list
  if (p(list.head()))
    [] + list.head() + filter(list.tail(), p)
  else
    filter(list.tail(), p)
}

l = filter(1..20, {n-> n % 2 == 0})
```

In the `filter()` method in Listing 4, I first check the size of the passed list and return it if it has no elements. Otherwise, I check the head of the list against my filtering predicate; if it passes, I add it to the list (with an initial empty list to make sure that I always return the correct type); otherwise, I recursively filter the tail.

The difference between Listing 3 and Listing 4 highlights an important question: Who's minding the state? In the imperative version, I am. I must create a new variable named `new_list`, I must add things to it, and I must return it when I'm done. In the recursive version, the language manages

the return value, building it up on the stack as the recursive return for each method invocation. Notice that every exit route of the `filter()` method in Listing 4 is a return call, which builds up the intermediate value on the stack.

Although not as dramatic a life improvement as garbage collection, this does illustrate an important trend in programming languages: offloading moving parts. If I'm never allowed to touch the intermediate results of the list, I cannot introduce bugs in the way that I interact with it.

This perspective shift about lists allows you explore other aspects, such as a list's size and scope.

Lazy lists in Groovy

One of the common features of functional languages is the *lazy list*: a list whose contents are generated only as you need it. Lazy lists allow you to defer initialization of expensive resources until you absolutely need them. They also allow the creation of infinite sequences: lists that have no upper bound. If you aren't required to say up front how big the list could be, you can let it be as big as it needs to be.

First, I'll show you an example of using a lazy list in Groovy in Listing 5, and then I'll show you the implementation:

Listing 5. Using lazy lists in Groovy

```
def prepend(val, closure) { new LazyList(val, closure) }

def integers(n) { prepend(n, { integers(n + 1) }) }

@Test
public void lazy_list_acts_like_a_list() {
  def naturalNumbers = integers(1)
  assertEquals('1 2 3 4 5 6 7 8 9 10',
      naturalNumbers.getHead(10).join(' '))
  def evenNumbers = naturalNumbers.filter { it % 2 == 0 }
  assertEquals('2 4 6 8 10 12 14 16 18 20',
      evenNumbers.getHead(10).join(' '))
}
```

The first method in Listing 5, `prepend()`, creates a new `LazyList`, allowing you to prepend values. The next method, `integers()`, returns a list of integers using the `prepend()` method. The two parameters I send to the `prepend()` method are the initial value of the list and a code block that includes code to generate the next value. The `integers()` method acts like a factory that returns the lazy list of integers with a value at the front and a way to calculate additional values in the rear.

To retrieve values from the list, I call the `getHead()` method, which returns the argument number of values from the top of the list. In Listing 5, `naturalNumbers` is a lazy sequence of all integers. To get some of them, I call the `getHead()` method, specifying how many integers I want. As the assertion indicates, I receive a list of the first 10 natural numbers. Using the `filter()` method, I retrieve a lazy list of even numbers and call the `getHead()` method to fetch the first 10 even numbers.

The implementation of `LazyList` appears in Listing 6:

Listing 6. `LazyList` implementation

```
class LazyList {
  private head, tail

  LazyList(head, tail) {
    this.head = head;
    this.tail = tail
  }

  def LazyList getTail() { tail ? tail() : null }

  def List getHead(n) {
    def valuesFromHead = [];
    def current = this
    n.times {
      valuesFromHead << current.head
      current = current.tail
    }
    valuesFromHead
  }

  def LazyList filter(Closure p) {
    if (p(head))
      p.owner.prepend(head, { getTail().filter(p) })
    else
      getTail().filter(p)
  }
}
```

A lazy list holds a head and tail, specified in the constructor. The `getTail()` method ensures that `tail` isn't null and executes it. The `getHead()` method gathers the elements that I want to return, one at a time, pulling the existing element off the head of the list and asking the tail to generate a new value. The call to `n.times {}` performs this operation for the number of elements requested, and the method returns the harvested values.

The `filter()` method in Listing 5 uses the same recursive approach as Listing 4 but implements it as part of the list rather than a stand-alone function.

Lazy lists exist in Java (see Resources) but are much easier to implement in languages that have functional features. Lazy lists work great in situations in which generating resources are expensive, such as getting lists of perfect numbers.

Lazy list of perfect numbers

If you've been following this article series, you're well aware of my favorite guinea-pig code, finding perfect numbers (see "Thinking functionally, Part 1"). One of the shortcomings of all the implementations so far is the need to specify the number for classification. Instead, I want a version that returns a lazy list of perfect numbers. Toward that goal, I've written a highly functional, very compact perfect-number finder that supports lazy lists, shown in Listing 7:

Listing 7. Pared-down version of number classifier, including `nextPerfectNumberFrom()` method

```
class NumberClassifier {
  static def factorsOf(number) {
      (1..number).findAll { i -> number % i == 0 }
  }

  static def isPerfect(number) {
      factorsOf(number).inject(0, {i, j -> i + j}) == 2 * number
  }

  static def nextPerfectNumberFrom(n) {
    while (! isPerfect(++n)) ;
    n
  }
}
```

If the code in the `factorsOf()` and `isPerfect()` methods seem obscure, you can see the derivation of those methods in the last installment. The new method, `nextPerfectNumber()`, uses the `isPerfect()` method to find the next perfect number beyond the number passed as the parameter. This method call will take a long time to execute even for small values (especially given how unoptimized this code is); there just aren't that many perfect numbers.

Using this new version of `NumberClassifier`, I can create a lazy list of perfect numbers, as shown in Listing 8:

Listing 8. Lazily initialized list of perfect numbers

```
def perfectNumbers(n) {
  prepend(n,
    { perfectNumbers(nextPerfectNumberFrom(n)) }) };

@Test
public void infinite_perfect_number_sequence() {
  def perfectNumbers = perfectNumbers(nextPerfectNumberFrom(1))
  assertEquals([6, 28, 496], perfectNumbers.getHead(3))
}
```

Using the `prepend()` method I defined in Listing 5, I construct a list of perfect numbers with the initial value as the head and a closure block that knows how to calculate the next perfect number as the tail. I initialize my list with the first perfect number after 1 (using a static import so that I can call my `NumberClassifier.nextPerfectNumberFrom()` method more easily), then I ask my list to return the first three perfect numbers.

Calculating new perfect numbers is expensive, so I would rather do it as little as possible. By building it as a lazy list, I can defer calculations until the optimum time.

It is more difficult to think about infinite sequences if your abstraction of "list" is "numbered slots." Thinking of a list as the "first element" and the "rest of the list" encourages you to think of the elements in the list rather than the structure, which in turn allows you to think about things like lazy lists.

Conclusion

One of the ways that developers make quantum leaps in productivity is by building effective abstractions to hide details. We would never get anywhere if we were still coding with ones and zeros. One of the appealing aspects of functional languages is the attempt to abstract more details away from developers. Modern dynamic languages on the JVM already give you many of these features. In this installment, I showed how just shifting your perspective a little bit on how lists are constructed allows the language to manage state during iteration. Also, when you think of lists as "head" and "tail," it allows you to build things like lazy lists and infinite sequences.

In the next installment, you'll see how Groovy metaprogramming can make your programs more functional — and lets you augment third-party functional libraries to make them work better in Groovy.

Resources

Learn

- *The Productive Programmer* (Neal Ford, O'Reilly Media, 2008): Neal Ford's most recent book discusses tools and practices that help you improve your coding efficiency.
- An Excursion in Java Recursion: Recursion in Java has well-known limits, including some pointed out in this blog.
- Find limit of recursion: Find the recursion depth on your platform with these tests from RosettaCode.
- Apache Commons LazyList: Apache Commons includes a lazy-list implementation.
- Browse the technology bookstore for books on these and other technical topics.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- Download IBM product evaluation versions or explore the online trials in the IBM SOA Sandbox and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

Discuss

- Get involved in the developerWorks community. Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.

About the author

Neal Ford

Neal Ford is a software architect and Meme Wrangler at **Thought**Works, a global IT consultancy. He also designs and develops applications, instructional materials, magazine articles, courseware, and video/DVD presentations, and he is the author or editor of books spanning a variety of technologies, including the most recent *The Productive Programmer*. He focuses on designing and building large-scale enterprise applications. He is also an internationally acclaimed speaker at developer conferences worldwide. Check out his Web site.