

*Charles Parnot*

# Replacing Photoshop With NSString

## HACKERMONTHLY

Issue 60 May 2015

**Curator**

Lim Cheng Soon

**Contributors**

Charles Parnot  
Caleb McDaniel  
Mary Rose Cook  
Oliver Hardt  
Michael Fogleman  
John Fuex  
Larry Gadea  
Chet Bolingbroke  
Paul Graham

**Proofreader**

Emily Griffin

**Printer**

Blurb

HACKER MONTHLY is the print magazine version of Hacker News — *news.ycombinator.com*, a social news website wildly popular among programmers and startup founders. The submission guidelines state that content can be “anything that gratifies one’s intellectual curiosity.” Every month, we select from the top voted articles on Hacker News and print them in magazine format. For more, visit *hackermonthly.com*

**Advertising**

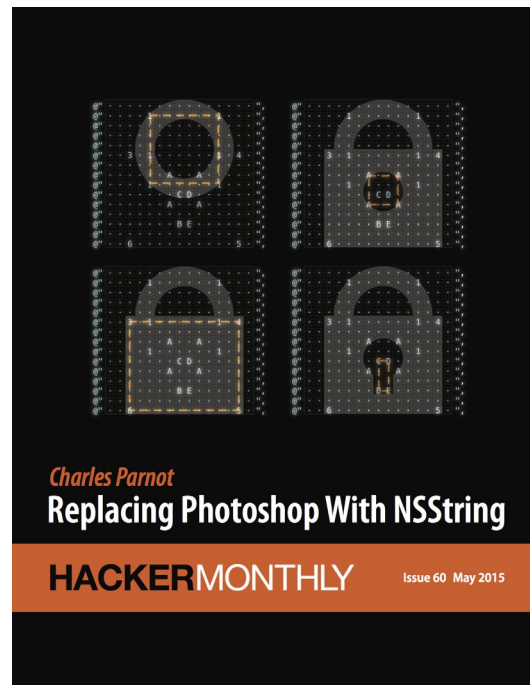
ads@hackermonthly.com

**Contact**

contact@hackermonthly.com

**Published by**

Netizens Media  
46, Taylor Road,  
11600 Penang,  
Malaysia.



# Contents

## FEATURES

### 04 **Replacing Photoshop With NSString**

*By* CHARLES PARNOT

### 10 **Why (and How) I Wrote My Academic Book in Plain Text**

*By* CALEB MCDANIEL

## PROGRAMMING

### 14 **A Practical Introduction to Functional Programming**

*By* MARY ROSE COOK

### 22 **Index Your Gmail Inbox with Elasticsearch**

*By* OLIVER HARDT

### 26 **I Made an NES Emulator**

*By* MICHAEL FOGLEMAN

### 28 **Programmers: Before You Turn 40, Get a Plan B**

*By* JOHN FUEX

### 30 **Run Your Own High-End Cloud Gaming Service on EC2**

*By* LARRY GADEA

## SPECIAL

### 32 **Shadowforge**

*By* CHET BOLINGBROKE

### 35 **What Doesn't Seem Like Work?**

*By* PAUL GRAHAM

# Replacing Photoshop With NSString

By CHARLES PARNOT

**A**N APP IS not just made of code. It also contains static assets like images and sounds. Images are typically created and edited with dedicated tools like Acorn (my favorite), Pixelmator, or the 800-pound gorilla, Photoshop. Ideally, the graphics are handled by an actual designer, which really is one of the best things we did for our app Findings [findingsapp.com]. But as a developer, it can be tedious to have to use a separate tool or involve another person, when all you need is a simple little icon with just a few straight lines, a square or a circle. Because of “retina,” you also have to create separate files for 1x, 2x, and now 3x-scale versions of the same drawing. Any small change or the addition of small variants can quickly become a cumbersome and error-prone endeavor.

What’s a developer to do? Write code! I don’t remember the first time I decided to draw an image directly in code, but that seemed like a good idea at the moment. From a developer’s perspective, it is very tempting. Why use Photoshop when you have the most flexible tool ever: code? Photoshop was

written in code, so whatever Photoshop is doing, code can do! Alas, in practice, this is only a reasonable approach for very simple graphics. And even then, it is not a straightforward task, and it is not quite the amount of fun I had naively hoped for. I will first show you an example of what it entails, but fear not, I also have an alternative fun solution right after that.

## Way too much code

As promised, here is an example of one of those first times I actually drew an image using Objective C. Brace yourself:

```
// chevron is defined by 3
// points, the angle is always
// 90 degrees
//
// A
//   #
//   #
//   B
//   #
//   #
//   C

CGFloat rightMargin = 12.0;
CGFloat chevronHeight = 9.0; //
then chevronWidth = chevron-
Height/2
```

```
CGFloat lineWidth = 2.0;
NSRect bounds = [self bounds];
NSPoint middle =
    CGPointMake(NSMaxX(bounds)-
        rightMargin-lineWidth/2.0,
        (NSMinY(bounds)+NSMaxY(bou
            nds))/2.0);
NSPoint top = middle;
top.x -= chevronHeight/2.0;
top.y += chevronHeight/2.0;
NSPoint bottom = top;
bottom.y -= chevronHeight;

// draw the chevron in grey
NSBezierPath *chevronPath =
    [NSBezierPath bezierPath];
[chevronPath
    setLineWidth:lineWidth];
[chevronPath setLineJoinStyle:N
    SMiterLineJoinStyle];
[chevronPath setLineCapStyle:NS
    ButtLineCapStyle];
[chevronPath moveToPoint:top];
[chevronPath
    lineToPoint:middle];
[chevronPath
    lineToPoint:bottom];
NSColor *chevronColor =
    [NSColor colorWithCalibrated-
        White:0.4 alpha:1.0];
[chevronColor set];
[chevronPath stroke];
```

Wow, that is a lot of code for just drawing two lines at a 90-degree angle! And that is not even including the actual `NSImage` code. It is nice that I can easily change the color and the size, and that I get 1x, 2x and 3x in one go. But was all this code really worth the trouble? After this first experience, I was not sold, but still used that approach in a few more occasions, where very simple graphics were needed. It got a little easier as I gained experience, and the invested time paid off, but I remained frustrated by the situation. After a while, though, I realized that the most interesting part of the code was actually the ASCII art I was using as a guide to my drawing code:

```
//      A
//      #
//      #
//      B      <-- I WANT TO WRITE JUST THAT,
//      #      NOT THE REST OF THE CODE!
//      #
//      C
```

This “drawing” described very nicely what I wanted to do, better than any comment I could ever write for any kind of code, in fact. That ASCII art was a great way to show directly in my code what image would be used in that part of the UI, without having to dig into the resources folder. The actual drawing code suddenly seemed superfluous. What if I could just pass the ASCII art into `NSImage` directly?

## ASCIIImage: combining ASCII art and Kindergarten skills

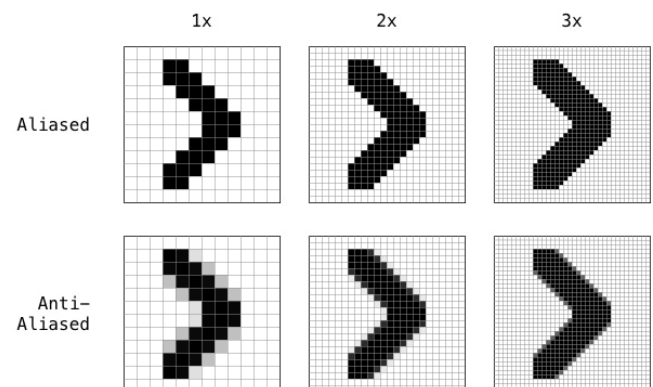
Xcode does not compile ASCII art, so I decided I would write the necessary “ASCII art compiler” myself. OK, I did not write a compiler, but a small fun project called “ASCIIImage”. It works on iOS and Mac as a simple `UIImage/NSImage` category with a couple of factory methods. It is open-source and released under the MIT license on GitHub. [hn.my/asciiimage] I also set up a landing page with a link to an editor hacked together by @mz2 in just a few hours during NSConference: [asciiimage.org](http://asciiimage.org).

It is very easy to use and has limited capabilities. It is not just a toy project, though. I have been using it in a real app for the past year: Findings. But whatever you do, here is a good rule of thumb: as soon as you feel limited by it, you should fire off Acorn [flyingmeat.com/acorn] instead, or better yet, contact a designer. [twitter.com/wrinklypea]

Here is how you would use `ASCIIImage`, to draw a 2-point-thick chevron:

```
+ (UIImage *)chevronImageWithColor:(UIColor *)
color
{
    NSArray *asciiRep =
    @[
        @". . . . .",
        @". . . 1 2 . . . . .",
        @". . . A # # . . . . .",
        @". . . # # # . . . . .",
        @". . . . # # # . . . . .",
        @". . . . . 9 # 3 . . . . .",
        @". . . . . 8 # 4 . . . . .",
        @". . . . # # # . . . . .",
        @". . . # # # . . . . .",
        @". . . 7 # # . . . . .",
        @". . . 6 5 . . . . .",
        @". . . . . . . . . . .",
    ];
    return [self imageWithASCIIRepresentation:asciiRep
        color:[UIColor blackColor]
        shouldAntialias:NO];
}
```

And below are the images that will be generated depending on the drawing environment:



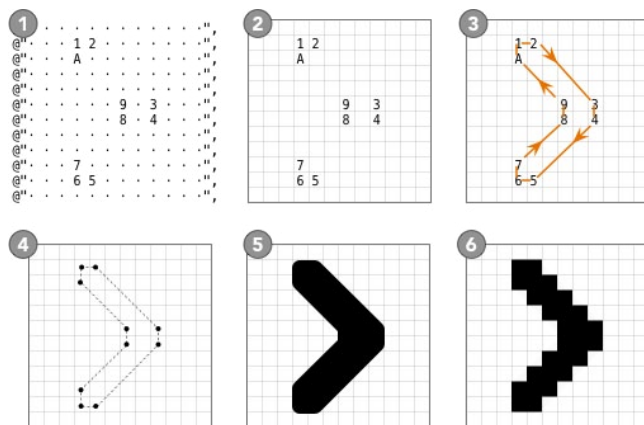
On iOS, the 1x/2x/3x versions will be generated based on the screen resolution of the device on which the app is running. On the Mac, the `ASCIIImage` implementation uses the `NSImage` block API, which means the drawing will happen at the right resolution the moment the image is rendered on screen. Note that I disabled anti-aliasing in the example code (so only the images on the top row will be generated as needed). For this kind of shape, the rendering is actually sharper and looks better without anti-aliasing.



Behind the scenes, ASCIIImage is doing simple, boring stuff. There are probably ways to make the parsing smarter and more user-friendly, but I just wanted things to work quickly without too much fuss and too much coding and debugging:

- It strips all whitespace; this is why all pixels need to be marked somehow (I chose the character “.” as the background in the example above);
- It checks consistency: all rows should have the same length;
- It parses the string to find digits and letters; everything else is ignored, namely the “.” and “#” characters in the example;
- Each digit/letter is assigned a corresponding NSPoint;
- It creates shapes based on the good old “Connect the Dots” technique you learnt in Kindergarten;
- Each shape is turned into NSBezierPath;
- Each Bezier path is rendered with the correct color and anti-aliasing flag

In the chevron example, there is just one shape, which is created and rendered as follows:

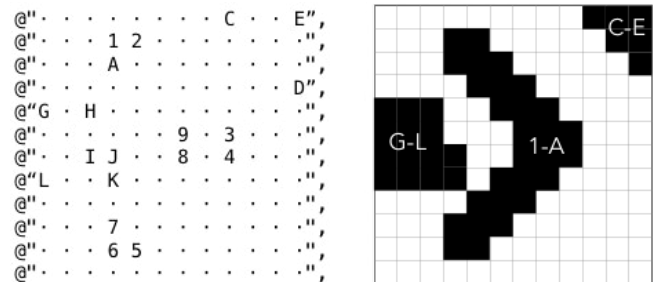


## Basics

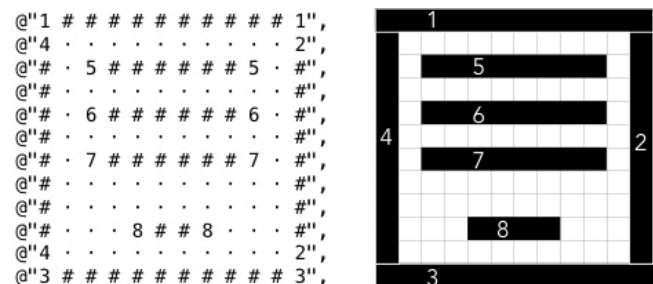
Here is a quick overview of ASCIIImage usage. The valid characters for connecting the dots are, in this order:

```
1 2 3 4 5 6 7 8 9 A B C D E F G H I J K L M N O
P Q R S T U V W X Y Z a b c d e f g h i j k l m
n p q r s t u v w x y z
```

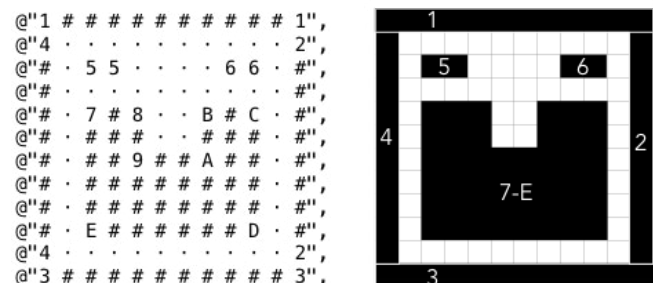
Each shape is defined by a series of sequential characters, and a new shape is started as soon as you skip a character in the above list. So the first shape could be defined by the series “123456”, then the next shape with “89ABCDEF”, the next with “HIJKLMNPO”, etc. The simplest method `+imageWithASCIIRepresentation:color:shouldAntialias:` will draw and fill each shape with the passed color (there is also a block-based method for more options). Here is an example with 3 shapes:



You can also draw straight lines by using the same character twice. In this case, you don’t need to skip a character before the next shape or line. Here is an example with a bunch of lines (remember, the “#” are only here as a visual guide for when you look at your code, but are ignored by ASCIIImage’s parser):

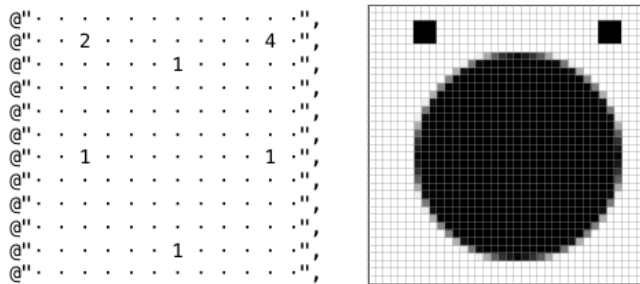


And you can combine shapes and lines, of course:

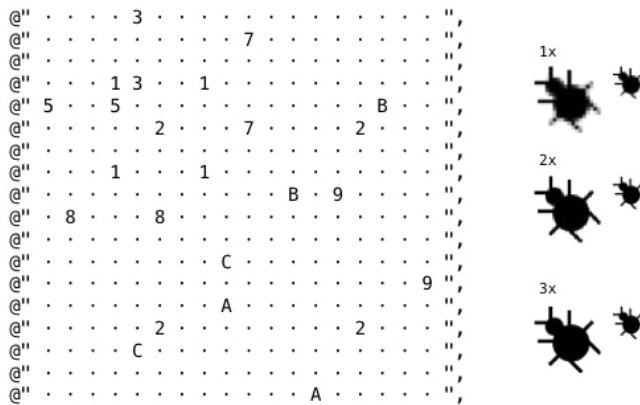


There are just 2 more special cases. You can create a single (square) pixel if you use an isolated character. And you can draw an ellipse by using the same character 3 or more times. The ellipse will be defined by the

largest enclosing rectangle for the points. If the rectangle is a square, the ellipse is a circle:



And finally, a more elaborate composition showing how far you can get with it. This particular ASCII art is entering obfuscation territory, which clearly defeats the purpose. The fun is still there, though!



That's it for the basics!

## Bells and whistles

There is a second factory method defined in `ASCIIImage`:

```
// This method offers more advanced options that
// can be set on each "shape", using the
// `contextHandler` block. The mutable
// dictionary passed by the block can be modified
// using the keys listed in the constants below.
// The dictionary initially contains the
// `ASCIIContextShapeIndex` key to
// signal which shape the context will be
// applied to.
+ (UIImage *)imageWithASCIIRepresentation:(NSArray *)rep
    contextHandler:(void (^)(NSMutableDictionary *ctx))handler;

/// keys for the dictionary context
extern NSString * const ASCIIContextShapeIndex;
```


```
extern NSString * const ASCIIContextFillColor;
extern NSString * const ASCIIContextStrokeColor;
extern NSString * const ASCIIContextLineWidth;
extern NSString * const ASCIIContextShouldClose;
extern NSString * const
ASCIIContextShouldAntialias;
```

This method allows you to apply different settings to the drawing of each element of the graphic. This is done via a mutable dictionary used as an argument in a block. Information goes both ways: from `ASCIIImage` to you, and then from you to `ASCIIImage`. You get the shape index (ordered based on the characters used in the ASCII art), and you set a stroke color, fill color, antialias flag, etc. Note that this context has not much in common with an actual `NSGraphicsContext`. It is very limited, and unfortunately, it is not possible to directly manipulate `NSGraphicsContext` for the kind of drawing `ASCIIImage` needs to do (or at least, there were enough gotchas that I decided against it).

Here is an example of how you could use the block-based method to layer multiple shapes on top of each other:

```
(UIImage *)deletionImage
{
    NSArray *asciiRep =
    @[
        @" . . . . 1 1 1 . . . .",
        @" . . 1 . . . . 1 . .",
        @" . 1 . . . . . . 1 .",
        @"1 . . 2 . . . 3 . 1",
        @"1 . . . # . # . . 1",
        @"1 . . . # . # . . 1",
        @"1 . . 3 . . . 2 . 1",
        @" . 1 . . . . . . 1 .",
        @" . . 1 . . . . 1 . .",
        @" . . . 1 1 1 1 1 . . .",
    ];
    return [UIImage imageWithASCIIRepresentation:
        :asciiRep
        contextHandler:^(NSMutableDictionary *context)
        {
            NSInteger index = [context[ASCIIContextShapeIndex] integerValue];
            if (index == 0)
            {
                context[ASCIIContextFillColor] =
```



And here is the result:



```
- (PARImage *)lockImage
{
```

1x  
2x  
3x

ASCII art obfuscation! The method name gives it away. Sort of. Here is how the string is parsed, shape after shape, layer after layer:



1x  

2x  

3x  



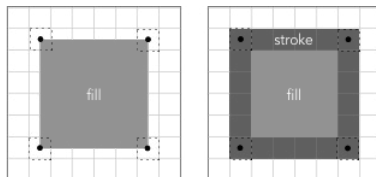
Again, not sure you'd want to go that far, but now you know you can!

## Tricky Bits

Implementing `ASCIIImage` was very straightforward, but there were still a few tricky bits:

- “Filling” out a shape actually involves both a fill and a stroke on `NSBezierPath`. To properly fill pixels and have proper pixel alignment, the vertices defining each Bezier path are in fact set to the middle of the 1x1-pt “pixel” represented in the ASCII art (1 x 1 pt ends up being 3 x 3 pixels in 3x scale for instance). When filling the path, the edges of the Bezier paths are thus drawn half-a-point away from the actual border. We then need to also apply a stroke of width 1-point, with the same color, to fill the full intended shape.

```
@". . . . .",
@". 1 # # # 2 .",
@". # . . . # .",
@". # . . . # .",
@". # . . . # .",
@". # . . . # .",
@". # . . . # .",
@". 4 # # # 3 .",
@". . . . .",
```



To really fill, you need to fill... and stroke.

- Without anti-aliasing, it is tricky to get the correct pixels to turn black. For this, I found that one should use a thicker line width for 45-degree lines, equal to the diagonal of a 1-pt square: the square root of 2. This width works fine for other angles, including horizontal and vertical lines, thus drawing of the lines is done using this width for aliased rendering, instead of the 1-pt width for anti-aliased rendering.
- For tests, one needs to trick the system into believing that the scale is 1x, 2x or 3x. On iOS, `ASCIIImage` has a special method with a scale argument, which is also used by the actual implementation (which simply passes the current device scale), ensuring that the same code path is in fact used. On OS X, it is trickier, in that the `NSImage` has to be rendered in a context where we control the “scale.” For this, the test actually renders the image returned by `ASCIIImage` into...another `NSImage`, with the correctly-scaled dimensions, so we get an artificial 1x context at a scaled-up size.

- The scaling on iOS and OS X is handled differently. On iOS, the Bezier paths need to be drawn directly at the right pixel size, and the Y axis is upside-down. On OS X, scaling is implicit, and drawing is done using points, not pixels. ■

---

After working as a biology scientist for 15 years, Charles Parnot switched to the life of a software developer. He is currently the developer and co-creator of the Findings app [findingsapp.com], an electronic lab notebook aimed at scientists and researchers. He is @cparnot on Twitter.

Reprinted with permission of the original author.  
First appeared in *hn.my/nsstring* (cocoamine.net)

# Why (and How) I Wrote My Academic Book in Plain Text

By CALEB MCDANIEL

**T**HESE DAYS, IT seems like the ancient past of personal computing is becoming the wave of the future. Do a simple search for writing in plain text and you'll find thousands of people making the case for using a file format (\*.txt) that worked long before Microsoft Word was a sparkle in Bill Gates's eye. In fact, I don't even have to make the general case for using plain text here; people like David Sparks, Michael Schechter, and Lincoln Mullen have already done the work for me.

Nonetheless, academic writers — and particularly historians — may well be skeptical about whether working in plain text can really work for them. Most of us still intend for our writing to end up on a printed page. We need the kinds of formatting — like bold-face and italics — that do not exist in a plain text file. Most of all, we need footnotes and all the bibliographic trappings that, “like the high whine of the dentist's drill,” assure the reader that we are serious professionals and have done our

homework. Journals and academic presses generally want our work to be submitted as Word documents, not as text files.

I'm writing this article partly to tell you that none of these are insuperable obstacles for the academic historian who wants to use plain text. In fact, I wrote the entirety of my academic book [hn.my/abook], forthcoming in early 2013, in plain text files. Before submitting the manuscript to my press, I converted all of my plain text files, complete with notes about what to italicize and where to place footnotes, to Microsoft Word documents using a simple program called Pandoc [pandoc.org], and the press never knew the difference. I've done the same thing now with a conference paper and journal article, too. It is possible to write academic publications in plain text, and in fact, Lincoln Mullen and I are working on a paper that will spell out how to do so in detail.

Of course, what is possible is not always desirable, and in this article

I want to focus on the specific, idiosyncratic reasons why I wanted (and still want) to write this way, using nothing more than a text editor and Pandoc. I'll briefly conclude with a rough primer on how I used Pandoc for my book, but the main point of this post is about why I did it, with no apologies for the fact that many of these reasons may not be compelling to everyone. They were, and remain, compelling to me.

## Why Not Just Use Word?

Unless your name is Ken Thompson and you've been happily typing away in Ed, the standard text editor, since 1971, you're not coming to plain text in a vacuum. I wasn't. I wrote my dissertation in Microsoft Word, and started writing my book in the same. So for me, the question of why I would want to use plain text occurred in a much different form: Why don't I want to keep using Word? And over time, that question became easier and easier to answer:

## 1 Microsoft Word eats my work.

Have you ever worked on a complicated section of a chapter or a footnote, only to have Word crash for some unspecified reason and erase your work? I have. This only has to happen once to get you thinking about alternatives. But let's just put it this way (and here I'm being charitable): in all my years of using Word, this happened more than once. In all the time I've used text editors, it's never happened, even once.

## 2 Microsoft Word distracts me.

Everyone has their own writing tics and pathologies. One of mine is adjusting the “look” of a word processing document. A full-featured processor like Word practically invites this — you can double-space, you can single-space — heck, you can 1.75 space. You can write in Palatino, or how about Courier — or what about Bookman Old Style? Is your page breaking at a funny spot? Why not adjust the margins, or the point size, or force a break at another point?

All valid questions, in their own way. But none of these questions have to do with the most important one of all: What do you want to say? On a rough writing day when I didn't want to confront that question, I often found myself wasting time on formatting, thinking that getting the right look would juice my thoughts. But my preferences about these things are fickle, so changing fonts or font sizes one day would usually lead me to change it all again on another. This is the vicious cycle that makes word processors stupid and inefficient: they confuse the process of typesetting

with the process of composition. And this has only gotten worse over time.

Another of my personal writing distractions is the constant temptation to read what I wrote at my last session before starting to write new material. Therein lies great danger. Yet Microsoft Word opens every document at the top “by product design.” There are workarounds, but none of them takes into account the extent of the temptation I face whenever I open a chapter. Yes, I could press SHIFT+F5, but why don't I just glance over the introduction again ... In moments of weakness, that's all it would take for me to waste a writing session tweaking instead of producing new material.

One solution to this kind of distraction is to break up a long chapter into chunks or sections, so that when I open the working file, I'm right where I need to be. The trouble is that combining Documents back together, and splitting them up again when needed, is not very easy. Text files, on the other hand, are different; almost any text editor has the ability to quickly combine large numbers of files into a new, complete file while leaving the chunks alone. The same task can be accomplished at the UNIX command line with commands like this:

```
cat chap1a chap1b chap1c >
chap1
cat chap1 chap2 chap3 chap4 >
dissertation
cat chap* > book
```

I find it less distracting to open only that section of a chapter or an article I need to be working on, and plain text makes that easy to do. And many editors allow me, by design, to open a file at precisely the point where I last left off

writing. When I'm ready to re-read my work, I now do so when the spirit is willing, instead of when the flesh is weak.

## 3 Microsoft Word is not very mobile.

Getting an iPod touch was one of the straws that broke Word's back for me. Having the Touch opened the possibility of writing anywhere, whenever the mood struck. But try to find an app that makes this easy to do with Word documents on an iPhone or iPad, and you'll find it's not as easy or as cheap as it sounds. More importantly, your choices of apps to work with are, at this moment, highly limited.

Plain text, on the other hand, travels well. Want an app that makes editing text on the iPhone or iPad easy and cheap? Want more than two or three feasible options? Here you go [hn.my/textapp]. Since switching to plain text, I've been able to work on my book by the pool, in the car, on the bus, and wherever else my elusive Muse showed up. And unlike many of the apps for working with Word documents, synchronization with my desktop computer through Dropbox has been easy and automatic.

## 4 Microsoft Word is not cheap.

The price tag for a Microsoft product is steep — and even steeper when you factor in the need to upgrade regularly so that your work will remain backwards- and forwards-compatible. Plain text, whether you're using Windows or Mac, is free, and free (or very cheap) text editors are abundant. Switching to other software is trivially easy.

But the price of the software isn't the only "cost" associated with Word. There are also associated storage costs, especially if you use something like Dropbox for backup and synchronization across multiple computers. The final copy of my dissertation is a 1.2MB file; the folder that contains all of my draft files for the project is over 20 times that size. The folder containing the plain text files for my final book manuscript, and all previous working drafts, is 4MB. That's not a huge difference, but if I multiply the difference across the multiple projects I have going at any one time — lecture notes, articles, conference papers, short essays, and so on — it can add up. By keeping file sizes lean, I've managed to stay well under my free 2GB limit on Dropbox and don't foresee having to upgrade anytime soon. (Yes, I'm that cheap.)

## Kicking the Word Habit

Those were some of my reasons for wanting a viable alternative to doing my academic writing primarily in Word. Perhaps others have other, still better reasons for wanting to kick the habit. But all of those reasons beg the question of whether it can be done. The short answer is yes.

For many years I clung to Word because I thought I needed it to do my work; when unhappy with it, I would try out simpler but still complex alternatives like Mellel [mellel.com] or Bean [bean-osx.com]. But one day I realized that my needs as an academic writer are often very minimal. To do most of my writing I basically need only three kinds of special formatting: emphasis, blockquotes, and footnotes. (There are exceptions,

of course, and cases where foreign character sets or symbols are needed; but I'm talking about 99% of my own work.) Adding each of those things is simple using Pandoc-flavored markdown.

In plain English, that means that when I want to italicize a word in plain text, I *\*emphasize\** it with asterisks. If I want a blockquote, I simply begin the paragraph with an arrow character (>). With footnotes, I have more options, but usually when I'm writing, I simply use a caret and brackets at the end of the paragraph, and put the footnote in the brackets.

This is easier to understand by looking at an example. The first two paragraphs of my book introduction looked like this in my text editor:

On April 14, 1865, hours before Abraham Lincoln sat down for the last time at Ford's Theater in Washington, D.C., the Boston abolitionist William Lloyd Garrison sat on a platform in Charleston, South Carolina. More than three decades before, Garrison had founded the *\*Liberator\**, a newspaper dedicated to agitating for universal, immediate slave emancipation. In 1833, he also helped found the American Anti-Slavery Society (AASS), a group devoted to the same goal. And by the time he went to Charleston, Garrison had served as the Society's president for over twenty years. Only in the last few, however, had emancipation changed from a despised, minority opinion to the official policy of federal armies in a cataclysmic civil war.

Now, with the war ending and a Constitutional amendment to abolish slavery awaiting ratification, Garrison had come to Fort Sumter to attend a flag-raising ceremony at the invitation of the government. Undoubtedly his emotions about the trip were difficult to express, and not only because he met recently emancipated slaves, one of whom pressed a ten dollar bill into his hand. Garrison's emotions were also stirred because he could now celebrate a country he had long regarded with deep disillusionment--even disgust.<sup>^</sup>[Henry Mayer, *\*All on Fire: William Lloyd Garrison and the Abolition of Slavery\** (New York: St. Martin's Griffin, 1998), 577-580. Garrison and Thompson told the story about the ten-dollar bill to Frederick W. Chesson. See the entry for June 10, 1867, in Frederick Chesson Diary, May 1867-April 1868, REAS, 11/15.]

I like writing this way, with footnotes tied directly to the text, but it's also possible to name long footnotes with descriptive titles and separate them from the paragraph, as in another paragraph of the introduction:

As historian C. A. Bayly and others have shown, the nineteenth century witnessed the birth of a nascent "international civil society" and the rise of transnational "networks of information and political advocacy which, though less obvious than the rising national and imperial state, [were] no less important." Abolitionists experienced these

realities in their everyday lives thanks to revolutions in transportation and communications technology that knit the Atlantic World together and astonished their contemporaries. By the end of Garrison's life, one American abolitionist marveled to the Irish abolitionist Richard Davis Webb about the incredible "wilderness of waters" over which their letters always crossed. Yet "our regular, constant, almost daily intercourse by steam mail-vessels" had led these distant friends to "accept it as a matter of course!"<sup>[^atlantic]</sup>

[^atlantic]: C. A. Bayly, *The Birth of the Modern World, 1780-1914* (Oxford, U.K.: Blackwell, 2004), 118; SMJr to RDW, March 26, 1871, BPL, Ms.B.1.6.11.9. On the transportation revolution in the United States during this same period, see Howe, *What Hath God Wrought*.<sup>\*</sup> For overarching surveys of the persistence of the Atlantic World into the nineteenth century, especially as a zone of cultural and economic exchange, see Donna Gabaccia, "A Long Atlantic in a Wider World," *Atlantic Studies* 1, no. 1 (2004): 1-27; Aaron Spencer Fogleman, "The Transformation of the Atlantic World, 1776-1867," *Atlantic Studies* 6, no. 1 (2009): 5-28; Jürgen Osterhammel and Niels P. Petersson, *Globalization: A Short History* (Princeton, N.J.: Princeton University Press, 2005), 57-80; José C. Moya, "Modernization, Modernity, and the Trans/

formation of the Atlantic World in the Nineteenth Century," in *The Atlantic in Global History, 1500-2000*,<sup>\*</sup> ed. Jorge Cañizares-Esguerra and Erik R. Seeman (Upper Saddle River, N.J.: Pearson, 2007), 179-98; and Thistlethwaite's still useful *The Anglo-American Connection*.<sup>\*</sup> [actual footnote continues ...]

After installing Pandoc and writing the entire book manuscript in this way, converting all of my chapters into a docx was as easy as issuing a single command in my Mac's Terminal program:

```
pandoc -s -o book.docx *.txt
```

Having done that, it was possible to open the manuscript, now complete with numbered footnotes, and adjust the formatting to the specifications given to me by my press (i.e., double-spaced, endnotes instead of footnotes, and so on). But it's also important that I only had to do that adjusting once. By modifying the default "styles" within this Word document and then saving that docx, I could also instruct Pandoc to use my correctly formatted file as a template for future document generation, like this:

```
pandoc -s --reference-docx=/Users/wcm1/lsu.docx -o book.docx *.txt
```

Learning to use Pandoc does take some time, to be sure. You can copy the plain text file I provided in the last paragraph and try it out yourself to see the kind of default output Pandoc produces. But my point here is that even its minimal functionality enables me to turn plain text files into \*.docx files with relative ease. And only a little bit

of reading and experimenting also enables me to distribute my academic work in a variety of formats. When sending my book manuscript to colleagues for review, for example, I was able to provide PDF, EPUB, or DOCX copies without changing the underlying plain text files at all. In short, I'm living proof that writing an academic book for history in plain text is possible. And having done it once, I'm not looking back. ■

---

Dr. W. Caleb McDaniel is assistant professor of history at Rice University. He is author of the award-winning book *The Problem of Democracy in the Age of Slavery: Garrisonian Abolitionists and Transatlantic Reform*,<sup>\*</sup> creator of the *@Every3Minutes* Twitter bot, and a general editor for *The Programming Historian* [programminghistorian.org]. You can follow him on Twitter *@wcaleb*

Reprinted with permission of the original author.  
First appeared in *hn.my/p/text* (wcm1.web.rice.edu)



# A Practical Introduction to Functional Programming

By MARY ROSE COOK

**M**ANY FUNCTIONAL PROGRAMMING articles teach abstract functional techniques. That is, composition, pipelining, and higher order functions.

This one is different. It shows examples of imperative, dysfunctional code that people write every day and translates these examples to a functional style.

The first section of the article takes short, data-transforming loops and translates them into functional maps and reduces. The second section takes longer loops, breaks them up into units, and makes each unit functional. The third section takes a loop that is a long series of successive data transformations and decomposes it into a functional pipeline.

The examples are in Python, because many people find Python easy to read. A number of the examples eschew pythonicity in order to demonstrate functional techniques common to many languages: map, reduce, pipeline.

## A guide rope

When people talk about functional programming, they mention a dizzying number of “functional” characteristics. They mention immutable data, first class functions and tail call optimization. These are language features that aid functional programming. They mention mapping, reducing, pipelining, recursing, currying, and the use of higher order functions. These are programming techniques used to write functional code. They mention parallelization, lazy evaluation, and determinism. These are advantageous properties of functional programs.

Ignore all that. Functional code is characterized by one thing: the absence of side effects. It doesn’t rely on data outside the current function, and it doesn’t change data that exists outside the current function. Every other “functional” thing can be derived from this property. Use it as a guide rope as you learn.

This is a non-functional function:

```
a = 0
def increment1():
    global a
    a += 1
```

This is a functional function:

```
def increment2(a):
    return a + 1
```

## Don’t iterate over lists. Use map and reduce.

### Map

Map takes a function and a collection of items. It makes a new, empty collection, runs the function on each item in the original collection, and inserts each return value into the new collection. It returns the new collection.

This is a simple map that takes a list of names and returns a list of the lengths of those names:

```
name_lengths = map(len, ["Mary", "Isla", "Sam"])

print name_lengths
# => [4, 4, 3]
```

This is a map that squares every number in the passed collection:

```
squares = map(lambda x: x * x, [0, 1, 2, 3, 4])
```

```
print squares
# => [0, 1, 4, 9, 16]
```

This map doesn't take a named function. It takes an anonymous, inlined function defined with `lambda`. The parameters of the `lambda` are defined to the left of the colon. The function body is defined to the right of the colon. The result of running the function body is (implicitly) returned.

The unfunctional code below takes a list of real names and replaces them with randomly assigned code names.

```
import random

names = ['Mary', 'Isla', 'Sam']
code_names = ['Mr. Pink', 'Mr. Orange', 'Mr. Blonde']

for i in range(len(names)):
    names[i] = random.choice(code_names)

print names
# => ['Mr. Blonde', 'Mr. Blonde', 'Mr. Blonde']
```

(As you can see, this algorithm can potentially assign the same secret code name to multiple secret agents. Hopefully, this won't be a source of confusion during the secret mission.)

This can be rewritten as a map:

```
import random

names = ['Mary', 'Isla', 'Sam']

secret_names = map(lambda x: random.choice(
    ['Mr. Pink',
     'Mr. Orange',
     'Mr. Blonde']),
    names)
```

**Exercise 1.** Try rewriting the code below as a map. It takes a list of real names and replaces them with code names produced using a more robust strategy.

```
names = ['Mary', 'Isla', 'Sam']

for i in range(len(names)):
    names[i] = hash(names[i])

print names
# => [6306819796133686941, 8135353348168144921,
-1228887169324443034]
```

(Hopefully, the secret agents will have good memories and won't forget each other's secret code names during the secret mission.)

My solution:

```
names = ['Mary', 'Isla', 'Sam']

secret_names = map(hash, names)
```

### Reduce

Reduce takes a function and a collection of items. It returns a value that is created by combining the items.

This is a simple reduce. It returns the sum of all the items in the collection.

```
sum = reduce(lambda a, x: a + x, [0, 1, 2, 3, 4])

print sum
# => 10
```

`x` is the current item being iterated over. `a` is the accumulator. It is the value returned by the execution of the `lambda` on the previous item. `reduce()` walks through the items. For each one, it runs the `lambda` on the current `a` and `x` and returns the result as the `a` of the next iteration.

What is `a` in the first iteration? There is no previous iteration result for it to pass along. `reduce()` uses the first item in the collection for `a` in the first iteration and starts iterating at the second item. That is, the first `x` is the second item.

This code counts how often the word 'Sam' appears in a list of strings:

```
sentences = ['Mary read a story to Sam and
Isla.',
             'Isla cuddled Sam.',
             'Sam chortled.']
```

```
sam_count = 0
for sentence in sentences:
    sam_count += sentence.count('Sam')
```

```
print sam_count
# => 3
```

This is the same code written as a reduce:

```
sentences = ['Mary read a story to Sam and
Isla.',
             'Isla cuddled Sam.',
             'Sam chortled.']
```

```
sam_count = reduce(lambda a, x: a +
                    x.count('Sam'),
                    sentences,
                    0)
```

How does this code come up with its initial a? The starting point for the number of incidences of 'Sam' cannot be 'Mary read a story to Sam and Isla.' The initial accumulator is specified with the third argument to reduce(). This allows the use of a value of a different type from the items in the collection.

Why are map and reduce better?

First, they are often one-liners.

Second, the important parts of the iteration — the collection, the operation and the return value — are always in the same places in every map and reduce.

Third, the code in a loop may affect variables defined before it or code that runs after it. By convention, maps and reduces are functional.

Fourth, map and reduce are elemental operations. Every time a person reads a for loop, they have to work through the logic line by line. There are few structural regularities they can use to create a scaffolding on which to hang their understanding of the code. In contrast, map and reduce are at once building blocks that can be combined into complex algorithms, and elements that the code reader can instantly understand and abstract in their mind. "Ah, this code is

transforming each item in this collection. It's throwing some of the transformations away. It's combining the remainder into a single output."

Fifth, map and reduce have many friends that provide useful, tweaked versions of their basic behavior. For example: filter, all, any and find.

**Exercise 2.** Try rewriting the code below using map, reduce, and filter. Filter takes a function and a collection. It returns a collection of every item for which the function returned True.

```
people = [{'name': 'Mary', 'height': 160},
          {'name': 'Isla', 'height': 80},
          {'name': 'Sam'}]
```

```
height_total = 0
height_count = 0
for person in people:
    if 'height' in person:
        height_total += person['height']
        height_count += 1
```

```
if height_count > 0:
    average_height = height_total / height_count
```

```
print average_height
# => 120
```

If this seems tricky, try not thinking about the operations on the data. Think of the states the data will go through, from the list of people dictionaries to the average height. Don't try and bundle multiple transformations together. Put each on a separate line and assign the result to a descriptively-named variable. Once the code works, condense it.

My solution:

```
people = [{'name': 'Mary', 'height': 160},
          {'name': 'Isla', 'height': 80},
          {'name': 'Sam'}]
```

```
heights = map(lambda x: x['height'],
               filter(lambda x: 'height' in x,
                     people))
```

```
if len(heights) > 0:
    from operator import add
    average_height = reduce(add, heights) /
    len(heights)
```

## Write declaratively, not imperatively

The program below runs a race between three cars. At each time step, each car may move forwards or it may stall. At each time step, the program prints out the paths of the cars so far. After five time steps, the race is over.

This is some sample output:

```
-
--
--

--
--
---

---
--
---

----
---
----

----
----
-----
```

This is the program:

```
from random import random

time = 5
car_positions = [1, 1, 1]

while time:
    # decrease time
    time -= 1

    print ''
    for i in range(len(car_positions)):
        # move car
        if random() > 0.3:
            car_positions[i] += 1

        # draw car
        print '-' * car_positions[i]
```

The code is written imperatively. A functional version would be declarative. It would describe what to do, rather than how to do it.

## Use functions

A program can be made more declarative by bundling pieces of the code into functions.

```
from random import random

def move_cars():
    for i, _ in enumerate(car_positions):
        if random() > 0.3:
            car_positions[i] += 1

def draw_car(car_position):
    print '-' * car_position

def run_step_of_race():
    global time
    time -= 1
    move_cars()

def draw():
    print ''
    for car_position in car_positions:
        draw_car(car_position)

time = 5
car_positions = [1, 1, 1]

while time:
    run_step_of_race()
    draw()
```

To understand this program, the reader just reads the main loop. “If there is time left, run a step of the race and draw. Check the time again.” If the reader wants to understand more about what it means to run a step of the race, or draw, they can read the code in those functions.

There are no comments anymore. The code describes itself.

Splitting code into functions is a great, low brain-power way to make code more readable.

This technique uses functions, but it uses them as sub-routines. They parcel up code. The code is not functional in the sense of the guide rope. The functions in the code use state that was not passed as arguments. They affect the code around them by changing external variables, rather than by returning values. To check what a function really does, the reader must read each line carefully. If they find an external variable, they must find its origin. They must see what other functions change that variable.

## Remove state

This is a functional version of the car race code:

```
from random import random

def move_cars(car_positions):
    return map(lambda x: x + 1 if random() > 0.3
               else x,
               car_positions)

def output_car(car_position):
    return '-' * car_position

def run_step_of_race(state):
    return {'time': state['time'] - 1,
            'car_positions': move_
cars(state['car_positions'])}

def draw(state):
    print ''
    print '\n'.join(map(output_car, state['car_
positions']))

def race(state):
    draw(state)
    if state['time']:
        race(run_step_of_race(state))

race({'time': 5,
      'car_positions': [1, 1, 1]})
```

The code is still split into functions, but the functions are functional. There are three signs of this. First, there are no longer any shared variables. `time` and `car_positions` get passed straight into `race()`. Second, functions take parameters. Third, no variables are instantiated inside functions. All data changes are done with return values. `race()` recurses<sup>3</sup> with the result of `run_step_of_race()`. Each time a step generates a new state, it is passed immediately into the next step.

Now, here are two functions, `zero()` and `one()`:

```
def zero(s):
    if s[0] == "0":
        return s[1:]

def one(s):
    if s[0] == "1":
        return s[1:]
```

`zero()` takes a string, `s`. If the first character is `'0'`, it returns the rest of the string. If it is not, it returns `None`, the default return value of Python functions. `one()` does the same, but for a first character of `'1'`.

Imagine a function called `rule_sequence()`. It takes a string and a list of rule functions of the form of `zero()` and `one()`. It calls the first rule on the string. Unless `None` is returned, it takes the return value and calls the second rule on it. Unless `None` is returned, it takes the return value and calls the third rule on it. And so forth. If any rule returns `None`, `rule_sequence()` stops and returns `None`. Otherwise, it returns the return value of the final rule.

This is some sample input and output:

```
print rule_sequence('0101', [zero, one, zero])
# => 1
```

```
print rule_sequence('0101', [zero, zero])
# => None
```

This is the imperative version of `rule_sequence()`:

```
def rule_sequence(s, rules):
    for rule in rules:
        s = rule(s)
        if s == None:
            break

    return s
```

**Exercise 3.** The code above uses a loop to do its work. Make it more declarative by rewriting it as a recursion.

My solution:

```
def rule_sequence(s, rules):
    if s == None or not rules:
        return s
    else:
        return rule_sequence(rules[0](s),
rules[1:])
```



## Use pipelines

In the previous section, some imperative loops were rewritten as recursions that called out to auxiliary functions. In this section, a different type of imperative loop will be rewritten using a technique called a pipeline.

The loop below performs transformations on dictionaries that hold the name, incorrect country of origin, and active status of some bands.

```
bands = [{'name': 'sunset rubdown', 'country': 'UK', 'active': False},
         {'name': 'women', 'country': 'Germany', 'active': False},
         {'name': 'a silver mt. zion', 'country': 'Spain', 'active': True}]

def format_bands(bands):
    for band in bands:
        band['country'] = 'Canada'
        band['name'] = band['name'].replace('.', '')
        band['name'] = band['name'].title()

format_bands(bands)

print bands
# => [{'name': 'Sunset Rubdown', 'active': False, 'country': 'Canada'},
#      {'name': 'Women', 'active': False, 'country': 'Canada' },
#      {'name': 'A Silver Mt Zion', 'active': True, 'country': 'Canada'}]
```

Worries are stirred by the name of the function. “format” is very vague. Upon closer inspection of the code, these worries begin to claw. Three things happen in the same loop. The ‘country’ key gets set to ‘Canada’. Punctuation is removed from the band name. The band name gets capitalized. It is hard to tell what the code is intended to do and whether it does what it appears to do. The code is hard to reuse, hard to test, and hard to parallelize.

Compare it with this:

```
print pipeline_each(bands,
                    [set_canada_as_country,
                     strip_punctuation_from_name,
                     capitalize_names])
```

This code is easy to understand. It gives the impression that the auxiliary functions are functional because they seem to be chained together. The output from the previous one comprises the input to the next. If they are functional, they are easy to verify. They are also easy to reuse, easy to test, and easy to parallelize.

The job of `pipeline_each()` is to pass the bands, one at a time, to a transformation function, like `set_canada_as_country()`. After the function has been applied to all the bands, `pipeline_each()` bundles up the transformed bands. Then it passes each one to the next function.

Let’s look at the transformation functions.

```
def assoc(_d, key, value):
    from copy import deepcopy
    d = deepcopy(_d)
    d[key] = value
    return d

def set_canada_as_country(band):
    return assoc(band, 'country', "Canada")

def strip_punctuation_from_name(band):
    return assoc(band, 'name', band['name'].replace('.', ''))

def capitalize_names(band):
    return assoc(band, 'name', band['name'].title())
```

Each one associates a key on a band with a new value. There is no easy way to do this without mutating the original band. `assoc()` solves this problem by using `deepcopy()` to produce a copy of the passed dictionary. Each transformation function makes its modification to the copy and returns that copy.

Everything seems fine. Band dictionary originals are protected from mutation when a key is associated with a new value. But there are two other potential mutations in the code above. In `strip_punctuation_from_name()`, the unpunctuated name is generated by calling `replace()` on the original name. In `capitalize_names()`, the capitalized name is generated by calling `title()` on the original name. If `replace()` and `title()` are not functional, `strip_punctuation_from_name()` and `capitalize_names()` are not functional.

Fortunately, `replace()` and `title()` do not mutate the strings they operate on. This is because strings are immutable in Python. When, for

example, `replace()` operates on a band name string, the original band name is copied and `replace()` is called on the copy. Phew.

This contrast between the mutability of strings and dictionaries in Python illustrates the appeal of languages like Clojure. The programmer need never think about whether they are mutating data. They aren't.

**Exercise 4.** Try to write the `pipeline_each` function. Think about the order of operations. The bands in the array are passed, one band at a time, to the first transformation function. The bands in the resulting array are passed, one band at a time, to the second transformation function. And so forth.

My solution:

```
def pipeline_each(data, fns):
    return reduce(lambda a, x: map(x, a),
                  fns,
                  data)
```

All three transformation functions boil down to making a change to a particular field on the passed band. `call()` can be used to abstract that. It takes a function to apply and the key of the value to apply it to.

```
set_canada_as_country = call(lambda x: 'Canada',
                             'country')
strip_punctuation_from_name = call(lambda x:
x.replace('.', ''), 'name')
capitalize_names = call(str.title, 'name')
```

```
print pipeline_each(bands,
                    [set_canada_as_country,
                     strip_punctuation_from_name,
                     capitalize_names])
```

Or, if we are willing to sacrifice readability for conciseness, just:

```
print pipeline_each(bands, [call(lambda x:
'Canada', 'country'),
                             call(lambda x:
x.replace('.', ''), 'name'),
                             call(str.title,
'name')])
```

The code for `call()`:

```
def assoc(_d, key, value):
    from copy import deepcopy
    d = deepcopy(_d)
    d[key] = value
    return d

def call(fn, key):
    def apply_fn(record):
        return assoc(record, key, fn(record.
get(key)))
    return apply_fn
```

There is a lot going on here. Let's take it piece by piece.

One. `call()` is a higher order function. A higher order function takes a function as an argument, or returns a function. Or, like `call()`, it does both.

Two. `apply_fn()` looks very similar to the three transformation functions. It takes a record (a band). It looks up the value at `record[key]`. It calls `fn` on that value. It assigns the result back to a copy of the record. It returns the copy.

Three. `call()` does not do any actual work. `apply_fn()`, when called, will do the work. In the example of using `pipeline_each()` above, one instance of `apply_fn()` will set 'country' to 'Canada' on a passed band. Another instance will capitalize the name of a passed band.

Four. When an `apply_fn()` instance is run, `fn` and `key` will not be in scope. They are neither arguments to `apply_fn()`, nor locals inside it. But they will still be accessible. When a function is defined, it saves references to the variables it closes over: those that were defined in a scope outside the function and that are used inside the function. When the function is run and its code references a variable, Python looks up the variable in the locals and in the arguments. If it doesn't find it there, it looks in the saved references to close over variables. This is where it will find `fn` and `key`.

Five. There is no mention of bands in the `call()` code. That is because `call()` could be used to generate pipeline functions for any program, regardless of topic. Functional programming is partly about building up a library of generic, reusable, composable functions.

Good job. Closures, higher order functions and variable scope all covered in the space of a few paragraphs. Have a nice glass of lemonade.

There is one more piece of band processing to do. That is to remove everything but the name and country. `extract_name_and_country()` can pull that information out:

```
def extract_name_and_country(band):
    plucked_band = {}
    plucked_band['name'] = band['name']
    plucked_band['country'] = band['country']
    return plucked_band
```

```
print pipeline_each(bands, [call(lambda x:
    'Canada', 'country'),
                             call(lambda x:
x.replace('.', ''), 'name'),
                             call(str.title,
    'name'),
                             extract_name_and_
country]])
```

```
# => [{'name': 'Sunset Rubdown', 'country':
'Canada'},
      {'name': 'Women', 'country': 'Canada'},
      {'name': 'A Silver Mt Zion', 'country':
'Canada'}]
```

`extract_name_and_country()` could have been written as a generic function called `pluck()`. `pluck()` would be used like this:

```
print pipeline_each(bands, [call(lambda x:
    'Canada', 'country'),
                             call(lambda x:
x.replace('.', ''), 'name'),
                             call(str.title,
    'name'),
                             pluck(['name',
    'country'])]])
```

**Exercise 5.** `pluck()` takes a list of keys to extract from each record. Try to write it. It will need to be a higher order function.

My solution:

```
def pluck(keys):
    def pluck_fn(record):
        return reduce(lambda a, x: assoc(a, x,
record[x]),
                      keys,
                      {})
    return pluck_fn
```

## What now?

Functional code coexists very well with code written in other styles. The transformations in this article can be applied to any code base in any language. Try applying them to your own code.

Think of Mary, Isla, and Sam. Turn iterations of lists into maps and reduces.

Think of the race. Break code into functions. Make those functions functional. Turn a loop that repeats a process into a recursion.

Think of the bands. Turn a sequence of operations into a pipeline. ■

---

Mary Rose Cook writes code, makes music, works at Hacker School and lives in New York City.

Reprinted with permission of the original author.  
First appeared in [hn.my/functional](http://hn.my/functional) (maryrosecCook.com)

# Index Your Gmail Inbox with Elasticsearch

By OLIVER HARDT

I RECENTLY LOOKED AT my Gmail inbox and noticed that I have well over 50k emails taking up about 12GB of space, but there is no good way to tell what emails take up space, who I sent them to, who emails me, etc.

The goal of this tutorial is to load an entire Gmail inbox into Elasticsearch using bulk indexing and then start querying the cluster to get a better picture of what's going on.

## Prerequisites

Set up Elasticsearch [hn.my/esinst] and make sure it's running at *http://localhost:9200*

I use Python and Tornado [tornadoweb.org] for the scripts to import and query the data. Run `pip install tornado chardet` to install Tornado and chardet.

## Alright, where do we start?

First, go here [hn.my/dlgmail] and download your Gmail mailbox. Depending on the amount of emails you have accumulated, this might take a while.

The downloaded archive is in the mbox format, and Python provides libraries to work with the mbox format, so that's easy.

The overall program will look something like this:

```
mbox = mailbox.UnixMailbox(open('emails.mbox',
'rb'), email.message_from_file)
```

```
for msg in mbox:
    item = convert_msg_to_json(msg)
    upload_item_to_es(item)
```

```
print "Done!"
```

## Ok, tell me more about the details

The full Python code is here: *hn.my/updatepy*

### Turn mbox into JSON

First, we've got to turn the mbox format messages into JSON so we can insert it into Elasticsearch. Here is some sample code [hn.my/mboxjson] that was very useful when it came to normalizing and cleaning up the data.

A good first step:

```
def convert_msg_to_json(msg):
    result = {'parts': []}
    for (k, v) in msg.items():
        result[k.lower()] = v.decode('utf-8',
'ignore')
```

Additionally, you also want to parse and normalize the From and To email addresses:

```
for k in ['to', 'cc', 'bcc']:
    if not result.get(k):
        continue
    emails_split = result[k].replace('\n', '').
replace('\t', '').replace('\r', '').replace(' ',
').encode('utf8').decode('utf-8', 'ignore').
split(',')
    result[k] = [ normalize_email(e) for e in
emails_split]

if "from" in result:
    result['from'] =
normalize_email(result['from'])
```

Elasticsearch expects timestamps to be in microseconds, so let's convert the date accordingly.

```
if "date" in result:
    tt = email.utils.parsedate_
tz(result['date'])
    result['date_ts'] = int(calendar.timegm(tt)
- tt[9]) * 1000
```

We also need to split up and normalize the labels.

```
labels = []
if "x-gmail-labels" in result:
    labels = [l.strip().lower() for l in
result["x-gmail-labels"].split(',')]
    del result["x-gmail-labels"]
result['labels'] = labels
```

Email size is also interesting, so let's break that out.

```
parts = json_msg.get("parts", [])
json_msg['content_size_total'] = 0
for part in parts:
    json_msg['content_size_total'] += len(part.
get('content', ""))
```

## Index the data with Elasticsearch

The simplest approach is a PUT request per item:

```
def upload_item_to_es(item):
    es_url = "http://localhost:9200/gmail/
email/%s" % (item['message-id'])
    request = HTTPRequest(es_url, method="PUT",
body=json.dumps(item), request_timeout=10)
    response = yield http_client.fetch(request)
    if not response.code in [200, 201]:
        print "\nfailed to add item %s" %
item['message-id']
```

However, Elasticsearch provides a better method for importing large chunks of data: bulk indexing. Instead of making one HTTP request per document and indexing individually, we batch them in chunks of 1000 or so documents and then index them.

Bulk messages are of the format:

```
cmd\n
doc\n
cmd\n
doc\n
...
```

where cmd is the control message for each doc we want to index. For our example, cmd would look like this:

```
cmd = {'index': {'_index': 'gmail', '_type':
'email', '_id': item['message-id']}}`
```

The final code looks something like this:

```
upload_data = list()
for msg in mbox:
    item = convert_msg_to_json(msg)
    upload_data.append(item)
    if len(upload_data) == 100:
        upload_batch(upload_data)
        upload_data = list()

if upload_data:
    upload_batch(upload_data)

and

def upload_batch(upload_data):

    upload_data_txt = ""
    for item in upload_data:
        cmd = {'index': {'_index': 'gmail', '_
type': 'email', '_id': item['message-id']}}
        upload_data_txt += json.dumps(cmd) +
"\n"
        upload_data_txt += json.dumps(item) +
"\n"

    request = HTTPRequest("http://local-
host:9200/_bulk", method="POST", body=upload_
data_txt, request_timeout=240)
    response = http_client.fetch(request)
    result = json.loads(response.body)
    if 'errors' in result:
        print result['errors']
```



## Ok, show me some data!

After indexing all your emails, we can start running queries.

### Filters

If you want to search for emails from the last 6 months, you can use the range filter and search for gte the current time (now) minus 6 months:

```
curl -XGET 'http://localhost:9200/gmail/email/_search?pretty' -d '{
  "filter": { "range" : { "date_ts" : { "gte":
    "now-6M" } } } }
'
```

or you can filter for all emails from 2014 by using gte and lt

```
curl -XGET 'http://localhost:9200/gmail/email/_search?pretty' -d '{
  "filter": { "range" : { "date_ts" : {
    "gte": "2013-01-01T00:00:00.000Z", "lt":
    "2014-01-01T00:00:00.000Z" } } } }
'
```

You can also quickly query for certain fields via the q parameter. This example shows you all your Amazon shipping info emails:

```
curl "localhost:9200/gmail/email/_search?pretty&q=from:ship-confirm@amazon.com"
```

### Aggregation queries

Aggregation queries let us bucket data by a given key and count the number of messages per bucket. For example, number of messages grouped by recipient:

```
curl -XGET 'http://localhost:9200/gmail/email/_search?pretty&search_type=count' -d '{
  "aggs": { "emails": { "terms" : { "field" : "to",
    "size": 10 }
  } } }
'
```

Result:

```
"aggregations" : {
  "emails" : {
    "buckets" : [ {
      "key" : "noreply@github.com",
      "doc_count" : 1920
    }, { "key" : "oliver@gmail.com",
      "doc_count" : 1326
    }, { "key" : "michael@gmail.com",
      "doc_count" : 263
    }, { "key" : "david@gmail.com",
      "doc_count" : 232
    }
    ...
  ]
}
```

This one gives us the number of emails per label:

```
curl -XGET 'http://localhost:9200/gmail/email/_search?pretty&search_type=count' -d '{
  "aggs": { "labels": { "terms" : { "field" :
    "labels", "size": 10 }
  } } }
'
```

Result:

```
"hits" : {
  "total" : 51794,
},
"aggregations" : {
  "labels" : {
    "buckets" : [ {
      "key" : "important",
      "doc_count" : 15430
    }, { "key" : "github",
      "doc_count" : 4928
    }, { "key" : "sent",
      "doc_count" : 4285
    }, { "key" : "unread",
      "doc_count" : 510
    }
    ...
  ]
}
```

Use a date histogram you can also count how many emails you sent and received per year:

```
curl -s "localhost:9200/gmail/email/_search?pretty&search_type=count" -d '{
  "aggs": {
    "years": {
      "date_histogram": {
        "field": "date_ts", "interval": "year"
      }
    }
  }
}
```

Result:

```
"aggregations" : {
  "years" : {
    "buckets" : [ {
      "key_as_string" :
"2004-01-01T00:00:00.000Z",
      "key" : 1072915200000,
      "doc_count" : 585
    }, {
      ...
    }, {
      "key_as_string" :
"2013-01-01T00:00:00.000Z",
      "key" : 1356998400000,
      "doc_count" : 12832
    }, {
      "key_as_string" :
"2014-01-01T00:00:00.000Z",
      "key" : 1388534400000,
      "doc_count" : 7283
    } ]
  }
}
```

Write aggregation queries to work out how much you spent on Amazon/Steam:

```
GET _search
{
  "query": {
    "match_all": {}
  },
  "size": 0,
  "aggs": {
    "group_by_company": {
      "terms": {
        "field": "order_details.merchant"
      },
      "aggs": {
        "total_spent": {
          "sum": {
            "field": "order_details.order_
total"
          }
        },
        "postage": {
          "sum": {
            "field": "order_details.post-
age"
          }
        }
      }
    }
  }
}
```

---

Oliver Hardt is a software engineer for Bitly where he scales distributed systems and makes the internet shorter, one link at a time.

Reprinted with permission of the original author.  
First appeared in [hn.my/els](http://hn.my/els)

# I Made an NES Emulator

*Here's What I Learned About the Original Nintendo*

By MICHAEL FOGLEMAN

I RECENTLY CREATED MY OWN NES emulator [hn.my/foglenes]. I did it mostly for fun and to learn about how the NES worked. I learned some interesting things, so I wrote this article to share. There is a lot of documentation already out there, so this is just meant to highlight some interesting tidbits.

## The CPU

The NES used the MOS 6502 (at 1.79 MHz) as its CPU. The 6502 is an 8-bit microprocessor that was designed in 1975. (Forty years ago!) This chip was very popular — it was also used in the Atari 2600 & 800, Apple I & II, Commodore 64, VIC-20, BBC Micro and more. In fact, a revision of the 6502 (the 65C02) is still in production today.

The 6502 had relatively few registers (A, X & Y) and they were special-purpose registers. However, its instructions had several addressing modes including a “zero page” mode that could reference the first 256 words (\$0000 — \$00FF) in memory. These opcodes required fewer bytes in program memory and fewer CPU cycles during execution. One way of looking at this is that a developer can treat

these 256 slots like “registers.”

The 6502 had no multiply or divide instructions. And, of course, no floating point. There was a BCD (Binary Coded Decimal) mode but this was disabled in the NES version of the chip — possibly due to patent concerns.

The 6502 had a 256-byte stack with no overflow detection.

The 6502 had 151 opcodes (of a possible 256). The remaining 105 values are illegal/undocumented opcodes. Many of them crash the processor. But some of them perform possibly useful results by coincidence. As such, many of these have been given names based on what they do.

The 6502 had at least one hardware bug, with indirect jumps. JMP (<addr>) would not work correctly if <addr> was of the form \$xxFF. When reading two bytes from the specified address, it would not carry the FF->00 overflow into the xx. For example, it would read \$10FF and \$1000 instead of \$10FF and \$1100.

## Memory Map

The 6502 had a 16-bit address space, so it could reference up to 64KB of memory. But the NES

had just 2KB of RAM at addresses \$0000 to \$0800. The rest of the address space was for accessing the PPU, the APU, the game cartridge, input devices, etc.

Some address lines were unwired, so large blocks of the address space actually mirror other addresses. For example, \$1000 to \$1800 mirrors the RAM at \$0000 to \$0800. Writing to \$1000 is equivalent to writing to \$0000.

## The PPU (Picture Processing Unit)

The PPU generated the video output for the NES. Unlike the CPU, the PPU chip was specially-built for the NES. The PPU ran at 3x the frequency of the CPU. Each cycle of the PPU output one pixel while rendering.

The PPU could render a background layer and up to 64 sprites. Sprites could be 8x8 or 8x16 pixels. The background could be scrolled in both the X and Y axis. It supported “fine” scrolling (one pixel at a time). This was kind of a big deal back then.

Both the background and sprites were made from 8x8 tiles. Pattern tables in the cartridge ROM defined these tiles. The patterns

only specified two bits of the color. The other two bits came from an attribute table. A nametable specified which tiles go where in the background. All in all, it seems convoluted compared to today's standards. I had to explain to my coworker that it wasn't "just a bitmap."

The background was made up of  $32 \times 30 = 960$  of these  $8 \times 8$  tiles. Scrolling was implemented by rendering more than one of these  $32 \times 30$  backgrounds, each with an offset. If scrolling in both the X and Y axis, up to four of these backgrounds could become visible. However, the NES only supported two, so different mirroring modes were used for horizontal or vertical mirroring.

The PPU contained 256 bytes of OAM — Object Attribute Memory — that stored the sprite attributes for all 64 sprites. The attributes include the X and Y coordinate of the sprite, the tile number for the sprite and a set of flags that specified two bits of the sprite's color, specified whether the sprite appears in front of or behind the background layer, and allowed flipping the sprite vertically and/or horizontally. The NES supported a DMA copy from the CPU to quickly copy a chunk of 256 bytes to the entire OAM. This direct access was about four times faster than manually copying the bytes.

Although the PPU supported 64 sprites, only 8 could be shown on a single scan line. An overflow flag would be set so that the program could handle a situation with too many sprites on one line. This is why the sprites flicker when there is a lot of stuff going on in the game. Also, there was a hardware bug that caused the overflow flag to sometimes not work properly.

Many games would make changes mid-frame so that the PPU would do one thing for one part of the screen and something else for the other — often used for split scrolling or rendering a score bar. This required precise timing and knowing exactly how many CPU cycles each instruction used. Things like this make emulation hard.

The PPU had a primitive form of collision detection — if the first (zeroth) sprite intersected the background, a flag would be set indicating a "sprite zero hit." Only one such hit could occur per frame.

The NES had a built-in palette of 54 distinct colors — these were the only colors available. It wasn't RGB; the colors in the palette basically spit out a particular chroma and luminance signal to the TV.

### The APU (Audio Processing Unit)

The APU supported two square wave channels, a triangle wave channel, a noise channel, and a delta modulation channel.

To play sounds, the game program would write to specific registers to configure these channels.

The square wave channels supported frequency and duration control, frequency sweeps, and volume envelopes.

The noise channel used a linear feedback shift register to generate pseudo-random noise.

The delta modulation channel could play samples from memory. The SMB3 music has a metal drum sound that used the DMC. TMNT3 had voices like "cowabunga" that used the DMC.

### Mappers

The address space reserved for the cartridge restricted games to 32KB of program memory and 8KB of

character memory (pattern tables). This was pretty limiting, so people got creative and implemented mappers.

A mapper is hardware on the cartridge itself that can perform bank switching to swap new program or character memory into the addressable memory space. The program could control this bank switching by writing to specific addresses that pointed to the mapper hardware.

Different game cartridges implemented this bank switching in different ways, so there are dozens of different mappers. Just as an emulator must emulate the NES hardware, it must also emulate the cartridge mappers. However, about 90% of all NES games use one of the six most common mappers.

### ROM Files

An .nes ROM file contains the program memory banks and character memory banks from the cartridge. It has a small header that specifies what mapper the game used and what video mirroring mode it used. It also specifies whether battery-backed RAM was present on the cartridge.

### Conclusion

I wrote my emulator in Go using OpenGL + GLFW for video and PortAudio for audio. The code is all on GitHub, so check it out: [hn.my/foglenes](https://github.com/foglenes) ■

---

Michael is obsessed with programming. If he's not doing it, he's probably thinking about it. His GitHub portfolio is overflowing with random side projects. Somehow, he still has time for other hobbies and interests, including gardening, running, space exploration, listening to 80s music and playing with his two little kids.

Reprinted with permission of the original author.  
First appeared in [hn.my/nnes](https://hn.my/nnes)

# Programmers: Before You Turn 40, Get a Plan B

By JOHN FUEX

## Welcome to geezer town, junior.

While researching my recent article, “Age discrimination and Programming Jobs,” I discovered a 1998 Op-Ed piece from The New York Times that cited some startling statistics from the NSF and Census bureau about the longevity of a software engineering career.

*[S]ix years after finishing college, 57 percent of computer science graduates are working as programmers; at 15 years the figure drops to 34 percent, and at 20 years — when most are still only in their early 40’s — it is down to 19 percent. In contrast, the figures for civil engineering are 61 percent, 52 percent, and 52 percent.*

I find the defensive tone of the article and the use of dubious sampling of only computer science graduates to support its conclusion undermines its credibility. In a lot of ways, the Government has been very slow to grok the software engineering trade. In this study it completely ignores the significant number of working programmers who either earned their degree in another discipline or never finished college.

Still, smart money seems to concur that the software engineer depreciates only slightly more slowly than the machine he or she toils behind as exemplified in this 1996 comment from Craig Barrett,

then President and Co-founder of Intel.

*The half-life of an engineer, software or hardware, is only a few years.*

Sure, the guy’s a suit, but more importantly he was (at the time) a 57 year old former engineer publicly reinforcing the discriminatory notion of expiration dates on other engineers. It’s scary as hell to think that such an influential industry insider thinks that a programming career is roughly the same as a professional basketball player’s.

## My take on the issue

Considerable accusatory ink has been dedicated to the age discrimination problem in technology, but I suspect it may be an inevitable consequence of the rapid pace of change that defines this field.

Consider the following:

- The market value of an employee is primarily determined by experience in technologies relevant to the employer.
- Software engineering reliably undergoes a major technology shift at least every 10 years.
- While a technology shift doesn’t completely negate the skills of veterans, it certainly levels the playing field for recent grads.

Now put yourself in the shoes of a prospective hiring manager using a newer technology like Ruby on Rails for which nobody other than David Heinemeier has more than about 5 years of experience. Sure, that extra 10 years of C++ experience is a positive differentiator for the veteran over the upstart with the same 3 years of Rails experience. All things equal you’d naturally hire the guy with more total experience.

However, all things are NOT equal. Those 10 years of C++ experience made the veteran candidate progressively more expensive as they leveraged the value of that experience in jobs requiring C++. The problem is that the marginal utility of that extra experience must exceed the marginal cost of hiring the veteran to justify paying the premium.

Herein is the source of the problem. The more irrelevant experience a candidate has, the more lopsided the utility/value equation becomes, and this presumes that the manager even has the luxury of paying extra to get that experience.

Even if the veteran prices himself competitively with a younger candidate, the hiring manager has to consider the implications of bringing in someone taking a big pay cut. Will they have morale issues from day one? Are they going to change their mind after a month that they



really do need that extra cash and leave? It's a sticky situation.

The unfortunate truth is that unlike other forms of discrimination that are more arbitrary and capricious, age discrimination can often be a result of objective and sound business justifications. I'm not trying to justify it as an acceptable practice, but just trying to describe the pickle it puts the manager in trying to make a sound business decision without compromising the ethical and legal obligations of the company.

### So what's your plan B?

Assuming you aren't fabulously wealthy, accepted to clown college, or the fatal victim of a Red-bull induced heart attack by 40, a mitigation strategy is in order. Here are some viable options.

#### Work for the one person who would never discriminate against you.

No. Not your mother. You! If you aren't the entrepreneurial type, consider a consultancy. For some reason that I don't completely get, a little gray hair and a smattering of experience in different technologies can create a beneficial bias for companies when they are renting brains instead of buying them outright. It may have something to do with the tendency for consultants to be vetted from higher up in the management chain where the silver foxes live.

#### Give in to the dark side and go into management.

I'd argue that a career in programming does precious little to prepare someone for management, but clearly management thinks that everyone including technologists harbors a deep longing to "graduate"

into their ranks. I think it a fallacy that no one would continue to design and build software for 20 years unless they had no ambition or growth potential. However, people like me that respect such dedication to the craft are in the minority. Maybe it is best to just stop fighting it, but consider the following before taking the plunge:

- Mid-level managers often make very little more, if not the same as high level engineers.
- It gets progressively harder to keep up with new technology because you don't work directly with it.
- Meetings, politics, and dealing with unrealistic requests will pretty much become your life.
- You may try to avoid it, but management-speak will creep into your vocabulary (did you notice my "paradigm" comment earlier?)
- Even when it isn't your fault, it's your fault.
- Even when you make it succeed, your team should get the credit.
- Being the wunderkind as a technologist is much easier to do in technology than management, you'll have to check your ego at the door.
- You will be forced to make decisions that affect people's personal life (pay, bonus, firing, etc.), and this is hard to stomach sometimes.
- It is very empowering, enjoyable to be able to set the agenda and sometimes say, "No. We ain't doing that shit."

- Computers are predictable, people are complicated. You will eventually fantasize about robot employees.
- Mentoring can be very rewarding, but also very challenging.

*The most difficult thing in the world is to know how to do a thing and to watch someone else do it wrong without comment.*

– Theodore H. White.

#### You've got a cash cow, milk that sucker!

I know you love programming because you like technology, so this may go against your very nature, but no one says you've got to jump every time some snot-nosed kid invents a new way to run byte-code. You have invested a lot of time and energy mastering the technology you use, and your experience differentiates you. Money follows scarcity, and snow-birding on an older technology, if you can stomach it, may just be the way to protect your earning potential. The industry turns on a dime, but is slow to retire proven technology. It is highly likely that you will still be able to earn some decent coin in the technology you know and love even after a few decades. ■

---

John Fuex is a software development manager and product manager with 25 years of experience spanning all phases of software development. He currently works in Austin Texas specializing in eDiscovery technology and process automation to enable federal agencies to more efficiently handle large data collections for litigation or administrative document reviews.

Reprinted with permission of the original author.  
First appeared in [hn.my/planb](http://hn.my/planb) ([improvingsoftware.com](http://improvingsoftware.com))

# Run Your Own High-End Cloud Gaming Service on EC2

*How to use EC2 GPU machines + Steam In-Home Streaming + a VPN to play AAA titles on a shitty laptop*

By LARRY GADEA

**Y**OU MIGHT HAVE tried a service like the now defunct OnLive. Though personally I've played and beat many AAA games on the service, it unfortunately a) had a very limited selection and b) is now gone. I also have a bunch of games on Steam that I've played using my eGPU. With the new Macbook though, I won't be able to continue my low-end-laptop but high-end gaming extravaganza since there's no Thunderbolt. So why am I not concerned? Steam recently introduced In-Home Streaming, which basically creates a mini-OnLive in your own home with all the same Steam games I played with my eGPU. But... let's do it over the Internet!

## Cost

Playing games this way is actually quite economical — especially when comparing to purchasing a full-on gaming rig. Here are the costs you'll need to consider:

- GPU Instance runs about \$0.11/hr (on a Spot instance, regularly around \$0.70/hr)
- Data transfer will be around \$0.09/GB, and at a sustained ~10mbit, it'll cost you \$0.41/hr (4.5GB/hr)
- Hard drive (EBS General Purpose SSD) storage of 100GB is \$12.00/mo, or a bit under \$0.02/hr

This comes out to around \$0.54/hr. Not bad: for the cost of a \$1000 gaming PC, you get ~1900 hours on much higher-end hardware!

## The catch?

This is all fun and games, but you need to make sure of two things:

1. You are within 40ms to the closest AWS datacenter and has GPU instances
2. You have at least a 10mbit connection and it's unmetered

## Setting it up

1. On AWS, create a new EC2 instance. Use these settings:
  - Base image should be **Microsoft Windows Server 2012 R2 Base** (since Windows still has all the best games)
  - Use a **g2.xlarge** instance (to get an NVIDIA GRID K520 graphics card). There is no point using any larger instances since all they do is just give you more GPUs you can't use.
  - Use a Spot instance — it's significantly cheaper (1/7th the regular cost) than regular instances
  - For storage, I recommend at least 100GB (so you can install lots of fancy games)
  - Also for storage if you're using spot instances, make sure your primary disk doesn't get deleted on termination

- For the Security Group, I'd recommend just adding type All traffic
  - Finally, for the key pair, create a new one since you'll need one for Windows (to retrieve the password)
2. Once your spot instance is assigned, use Microsoft Remote Desktop to connect to it. The username is `Administrator` and the password you'll need to retrieve from the EC2 Console. Once inside, make sure to install TightVNC server and use Screen Sharing (or alternatively Screens which has better clipboard handling) to connect to the server. VNC is necessary so that the server uses the proper video card for rendering.
  3. Install the NVIDIA K520 drivers from the Nvidia website
  4. In order to make it actually use the video card, you'll need to completely remove the default driver. Open up Device Manager, and a) disable the `Microsoft Basic Display Adapter`, b) uninstall it and c) delete the driver file `C:\Windows\System32\Drivers\BasicDisplay.sys`. Reboot and VNC back in.
  5. Start the Windows Audio Service as per the instructions here [hn.my/winaud]. Also, since you're on EC2, those machines do not virtualize a sound card. Install VB-Cable so you can get sound. Alternatively, you can install the more commercial Razer Surround to simulate 5.1 — it's pretty cool.
  6. Install OpenVPN via the instructions here [hn.my/openvpnwin]. Make sure to use the TAP interface so Steam's multicast discovery gets forwarded. I personally use TunnelBlick [hn.my/tunnelblick] on my Mac as the client. You can alternatively use Hamachi [hn.my/hamachi] for both the server and client which is easier to set up, but I prefer to use non-commercial products. I was unable to get the built-in Windows VPN to work with multicast.
  7. Install Steam and get yourself on the Beta channel (available in the preferences). Also, start downloading whatever games you'll want to stream. On your own Steam installation, make sure to turn on Hardware Decoding in the Steam settings. I also recommend turning on Display Performance Information.
  8. Start gamin!

## Performance

While playing, make sure to hit F6 to see the latency graph. Anything above 50ms will make the delay somewhat noticeable, though I've played with delays up to 100ms. It just takes some getting used to and before you know it you won't even know you're streaming your games from a computer far, far away.

One other thing you should do is hit F8 while playing (note that sometimes this will cause the client to crash, but the file will still get written). F6 will do a dump of stats in the `C:\Program Files (x86)\Steam\logs\streaming_log.txt` file on the server. Open it up to see detailed latency timings. Here's an example of the interesting lines:

```
"AvgPingMS"
"11.066625595092773"

"AvgCaptureMS"
"4.555628776550293"

"AvgConvertMS"
"0.023172963410615921"

"AvgEncodeMS"
"5.5545558929443359"

"AvgNetworkMS"
"7.0888233184814453"

"AvgDecodeMS"
"3.7478375434875488"

"AvgDisplayMS"
"6.3670969009399414"

"AvgFrameMS"
"27.798770904541016"

"AvgFPS"
"57.622333526611328"
```

Unfortunately Steam doesn't support pulling the video from the H264 encoder on the GRID's NvFBC (which would reduce AvgEncodeMS a bunch). If you were running a GTX video card locally, this is one thing that'd make it faster than using EC2 (in addition to largely decreasing NetworkMS).

## Summary

If you have a) a fast internet connection and b) you're near an AWS datacenter with GPU instances, in my opinion, this is actually quite practical. Not only performance-wise, but it's also quite economical.

Happy gaming! ■

---

Larry is the founder of Envoy, an iPad-based visitor logbook for businesses. To blow off steam, Larry also loves tinkering with other bleeding-edge things like cloud gaming. He gets 6ms to the closest AWS datacenter.

---

Reprinted with permission of the original author.  
First appeared in [hn.my/cloudgaming](http://hn.my/cloudgaming) (lg.io)

# Shadowforge

## *John Carmack's First Game*

By CHET BOLINGBROKE

### Shadowforge

John D. Carmack  
(developer); Nite Owl  
Productions (publisher)  
Released 1989 or 1990  
for the Apple II.

Date Started: 4 April 2015

Date Ended: 4 April 2015

Total Hours: 2

Reload Count: 0

Difficulty: Easy (2/5)

Final Rating: 20

Ranking at Time of

Posting: 42/181 (23%)

**I**N PREPARATION FOR this posting, I read the first few chapters of David Kushner's *Masters of Doom*, and I was struck by the similarities I found between me and John Carmack. We're both about the same age, both nerdy and introverted as youths, more at home in front of computers than with other people. Our parents were both divorced at about the same age. We both experimented with burglary as teenagers (he was caught; I wasn't). We both got horrible grades in high school despite having the intelligence to do better. In our late teens, we both tried to break out of our

"nerd" roles by investing more in physical fitness (Carmack studied judo; I joined the Army Reserves). And we both dropped out of college, made some of the most iconic video games of the 1990s, and became multimillionaires. Okay, that last part may have just been him.

From Kushner's account, Carmack got out of a year in juvenile detention in 1986 or 1987, was given an Apple II by his parents, and got to work on *Shadowforge*, his first game. Although admittedly based on the look and feel of the early *Ultima* titles, he programmed it from scratch and sold

the completed game to Night Owl Productions, "a mom 'n' pop publisher that made most of its income from manufacturing camera batteries," for \$1000. He used the money to purchase an Apple IIgs and used it to write his second game, *Wraith: The Devil's Demise*, after he'd dropped out of the University of Kansas. He

used his developing programming skills to get a contract with Softdisk of Shreveport, Louisiana, and the result was the *Dark Designs* trilogy.

We, of course, have already had a look at *Dark Designs I* and *Dark Designs II*, both released in 1990. But some production issues at Night Owl also delayed the release of his first two games until 1989 and 1990. I naturally should have played them first. Rather than compound the error now by looking only at *Wraith*, I decided to reach back to 1989 and call up *Shadowforge* first.





Shadowforge feels like exactly what it is: a first game from a teenaged developer who grew up schooled on Ultima. It's so small that the only disk image I've been able to find also has half a dozen other games on it.

The game takes place in the town of Jaterus, which is being threatened by an evil mage named Greymere Shadowsender. Greymere's newly-constructed Shadowforge has given him unprecedented power, and the town needs a hero to descend into Greymere's three-level dungeon and destroy the device.

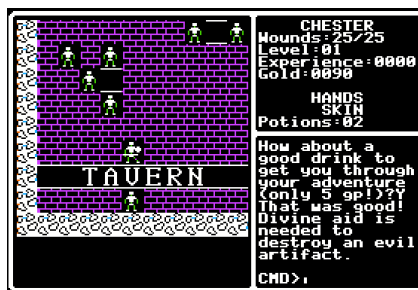


A dungeon scene from Shadowforge. I'm about to fire a bow at one of two enemies.

There's no character creation except designating a name. Each adventurer starts with 25 hit points, 0 experience, 100 gold, two potions, and has only his hands and skin for defense. Jaterus has an armorer, a weaponsmith, a tavern, a bowyer, an inn, a temple selling healing potions, and a casino hidden behind a secret door. You can bet 50 gold pieces on craps there; odds seem about 50/50.

There are miscellaneous NPCs running all over town, and one key difference between this game and Ultima is that you can't talk to any of them. You can't attack them, either; they really serve no purpose at all. The only "dialogue," as such, comes from tipping the bartender,

who provides a handful of hints for the quest ahead.



The armorer, weaponsmith, and bowyer each offer 3 or 4 items escalating in price and quality. As you enter the dungeon (which is right off the city; there's no outdoor area), you start to encounter goblins, ogres, and such. Killing them gives you experience and gold, which you spend on better equipment and a stock of healing potions.



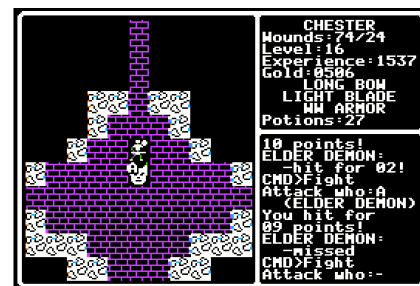
That's about all there is to it. At first, your expeditions to the dungeon are short, but once you get the best equipment and can carry more than a dozen potions at a time, they last a lot longer. Cleared rooms remain clear while you're still in the dungeon, but they respawn when you leave and return.

None of the three levels is terribly large. Although there are no special encounters or treasures to find, Carmack does make use of the walls and textures to create "scenes," often with large letters giving some kind of room title like LABORATORY or GOBLIN BAR-RACKS. This shows a clear Ultima II influence.



You get a new level for every 100 experience points, and each one comes with another 3 or 4 maximum hit points. Resting in the hotel restores maximum hit points; potions convey only 1-12 per gulp.

Combat consists of hitting (S)hoot if you see enemies from a distance and (F)ight if they're adjacent to you. There aren't many tactics except to take care that you don't blunder into foes. You can make some limited use of the terrain to make sure you don't get attacked by more than one foe at once. Foes that have missile weapons have no melee capability, so the best approach to them is to close the distance and start whacking. There is no magic in the game.



I fight an elder demon in melee combat on the way to the Shadowforge.

There are a few secret doors in the dungeon, signaled by subtle breaks in the wall pattern. Behind these, you can find special encounters with "merchants" who provide special items. I got a suit of "water walking" armor this way, along with a "light blade." I needed the former to get to the stairs from Level 2 to



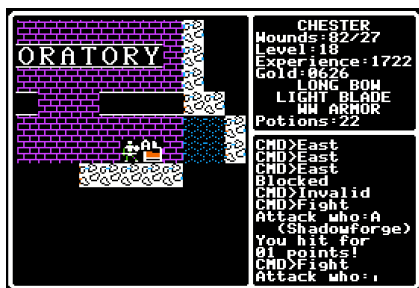
Level 3, and the latter to destroy the Shadowforge. There was apparently a magic bow somewhere, but I didn't find it.



The introductory text warns you not to confront Greymere directly, "since he can kill even an experienced adventurer with only a few spells," but when I ran into him on the third level, I was able to kill him in a few hits.



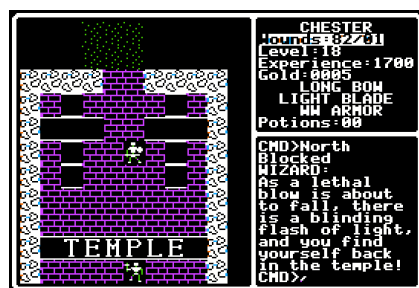
That kind of rendered the rest of the quest moot, I thought, but I kept exploring until I found the Shadowforge and hacked it to destruction.



In my version of the game, the endgame text shilled Wraith, meaning this is either a slightly later version or Night Owl didn't publish the original until they had Wraith in hand.



Overall, it was pretty easy. I didn't die once, and it took less than two hours to win. The game does allow you to save, and it autosaves every time you enter a new area. Death has you resurrected in the town's temple with a slight loss in experience, only 5 gold pieces, and no potions.



It's a promising game, certainly impressive for someone who was in his mid-teens when he wrote it. It showed that he was capable of whipping up a functional game engine that could serve as a basis for a more complicated experience, which he essentially offered in Wraith. Compared to other 1989-1990 games, particularly commercial titles, it doesn't offer much. It gets 1s, 2s, and 3s across the board in the GIMLET — its best categories are "Economy," "Interface," and "Gameplay" — culminating in a total score of 20. ■

Chet Bolingbroke is a blogger working his way through 40 years of computer role-playing games on his blog, "The CRPG Addict." He is 42 years old and lives in Salem, Massachusetts with his patient wife, Irene.

Reprinted with permission of the original author.  
First appeared in [hn.my/shadowforge](http://hn.my/shadowforge) (crpgaddict.blogspot.jp)

# What Doesn't Seem Like Work?

By PAUL GRAHAM

**M**Y FATHER IS a mathematician. For most of my childhood he worked for Westinghouse, modelling nuclear reactors.

He was one of those lucky people who know early on what they want to do. When you talk to him about his childhood, there's a clear watershed at about age 12, when he "got interested in maths." He grew up in the small Welsh seacoast town of Pwllheli. As we retraced his walk to school on Google Street View, he said that it had been nice growing up in the country.

"Didn't it get boring when you got to be about 15?" I asked.

"No," he said, "by then I was interested in maths."

In another conversation he told me that what he really liked was solving problems. To me the exercises at the end of each chapter in a math textbook represent work, or at best a way to reinforce what you learned in that chapter. To him the problems were the reward. The text of each chapter was just some advice about solving them. He said that as soon as he got a new

textbook he'd immediately work out all the problems — to the slight annoyance of his teacher, since the class was supposed to work through the book gradually.

Few people know so early or so certainly what they want to work on. But talking to my father reminded me of a heuristic the rest of us can use. If something that seems like work to other people doesn't seem like work to you, that's something you're well suited for. For example, a lot of programmers I know, including me, actually like debugging. It's not something people tend to volunteer; one likes it the way one likes popping zits. But you may have to like debugging to like programming, considering the degree to which programming consists of it.

The stranger your tastes seem to other people, the stronger evidence they probably are of what you should do. When I was in college I used to write papers for my friends. It was quite interesting to write a paper for a class I wasn't taking. Plus they were always so relieved.

It seemed curious that the same task could be painful to one person and pleasant to another, but I didn't realize at the time what this imbalance implied, because I wasn't looking for it. I didn't realize how hard it can be to decide what you should work on, and that you sometimes have to figure it out from subtle clues, like a detective solving a case in a mystery novel. So I bet it would help a lot of people to ask themselves about this explicitly. What seems like work to other people that doesn't seem like work to you? ■

---

Paul Graham is a programmer, writer, and investor. He co-founded Viaweb and Y Combinator. He is the author of *On Lisp* (1993), *ANSI Common Lisp* (1995), and *Hackers & Painters* (2004).

Reprinted with permission of the original author.  
First appeared in [hn.my/work](http://hn.my/work) ([paulgraham.com](http://paulgraham.com))

# Join the DuckDuckGo Open Source Community.



Create Instant Answers  
or share ideas and help  
change the future of search.

Featured IA: Regex Contributor: mintsoft  
Get started at [duckduckhack.com](https://duckduckhack.com)

The screenshot shows a DuckDuckGo search interface. The search bar contains 'regex cheat sheet'. Below the search bar, there are tabs for 'Answer', 'Images', and 'Videos'. The 'Answer' tab is selected, displaying a list of links and a snippet of text. The snippet is titled 'RegExLib.com Regular Expression Cheat Sheet (.NET Framework)' and includes a brief description of the cheat sheet's content. The snippet also mentions 'RegExLib.com Regular Expression Cheat Sheet (.NET) Metacharacters Defined; MChar Definition ^ Start of a string. \$ End of a ... see Regular Expression Options. [aeiou] Matches any single character included in the specified set of characters. [^aeiou] Matches any single character not in the ...' and provides a link to 'regexlib.com/CheatSheet.aspx'.

regex cheat sheet

Answer | Images | Videos

**RegExLib.com Regular Expression Cheat Sheet (.NET Framework)**

RegExLib.com Regular Expression **Cheat Sheet** (.NET) Metacharacters Defined; MChar Definition ^ Start of a string. \$ End of a ... see Regular Expression Options. [aeiou] Matches any single character included in the specified set of characters. [^aeiou] Matches any single character not in the ...

[regexlib.com/CheatSheet.aspx](https://regexlib.com/CheatSheet.aspx)



## Metrics and monitoring for people who know what they want

We know from experience that monitoring your servers and applications can be painful, so we built the sort of service that we would want to use. Simple to set up, responsive support from people who know what they're talking about, and reliably fast metric collection and dashboards.



**Dashboards**



**StatsD**



**Happiness**

**Now with Grafana!**

### Why Hosted Graphite?

- **Hosted metrics and StatsD:** Metric aggregation without the setup headaches
- **High-resolution data:** See everything like some glorious mantis shrimp / eagle hybrid\*
- **Flexible:** Lots of sample code, available on Heroku
- **Transparent pricing:** Pay for metrics, not data or servers
- **World-class support:** We want you to be happy!

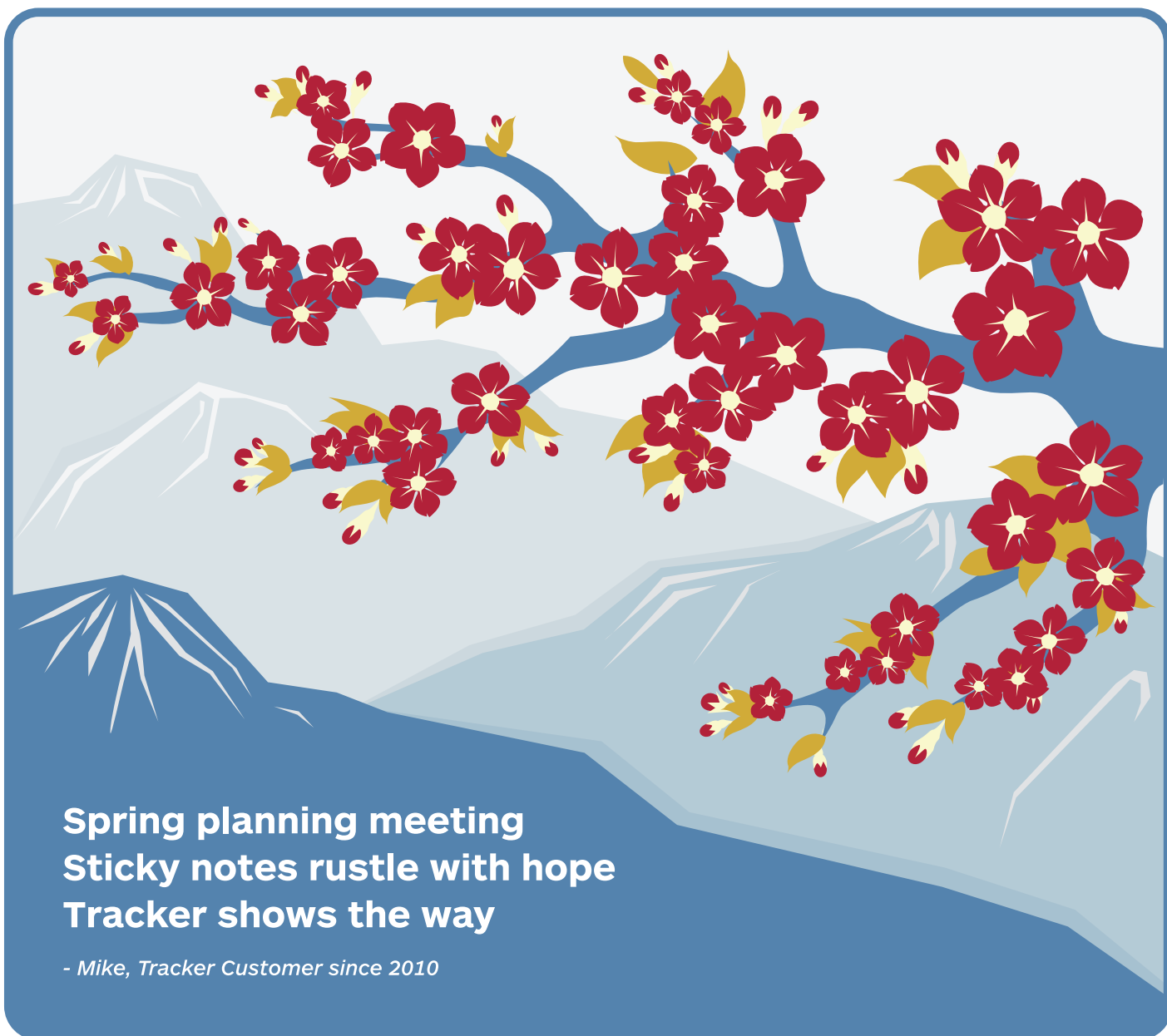
Promo code: **HACKER**

Grab a free trial at <http://www.hostedgraphite.com>

\*Hosted Graphite's mantis shrimp / eagle breeding program has been unsuccessful thus far



**HOSTEDGRAPHITE**



## Spring planning meeting Sticky notes rustle with hope Tracker shows the way

- Mike, Tracker Customer since 2010

## Discover the newly redesigned **Pivotal Tracker**

As our customers know too well, building software is challenging. That's why we created Pivotal Tracker, a pleasure-to-use project management tool, designed to facilitate constructive communication, keep teams focused, and reflect the true status of all your software projects.

With a new UI, cross-project functionality, in-app notifications and more, staying zen in the face of looming business deadlines just got a little easier.

Sign up for a free trial, no credit card required, at [pivotaltracker.com](https://pivotaltracker.com).