Why Free Plans Don't Work

HACKERMONTHLY Issue 5 October 2010

Curator
Lim Cheng Soon

Contributors
Shai Simonson
Fernando Gouvêa
Ruben Gamez
Steve Yegge
Tom Preston-Werner
Slava Akhmechet
Dave Ward
Sym Kat
Ben Pieratt
David Kadavy
Alexis Ohanian

Proofreader
Ricky de Laveaga

Illustrators
Jaime G. Wong
Pasquale D'Silva

Printer
MagCloud

E-book Conversion
Fifobooks.com

Advertising
ads@hackermonthly.com

Contact
contact@hackermonthly.com

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

Curator's Note

I had to put Ruben Gamez's excellent "Why Free Plans Don't Work" on the cover (with another great illustration by Pasquale D'Silva). A lot of people have been comparing Hacker Monthly to Wired (or the old Wired), and Ruben's article is the antithesis of Wired Issue 16.03's cover story: "Free! Why $0.00 is the future of business."

This issue is light on the "Startup" section and heavy on programming. Along with the featured "How To Read Math," there's a long article by Steve Yegge on why compilers matter, plus articles on LaTeX, SSH, jQuery, and more.

Oh, and make sure you don't miss the awesome advertisement by Breadpig.

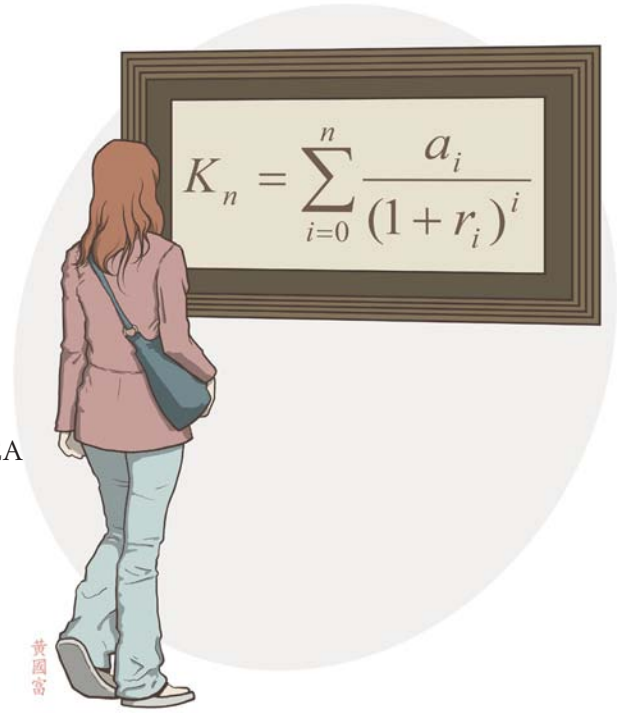
I hope you enjoy reading this issue as much as I enjoyed curating it. ■

— Lim Cheng Soon

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*.

Cover Illustration: Pasquale D'Silva

Contents

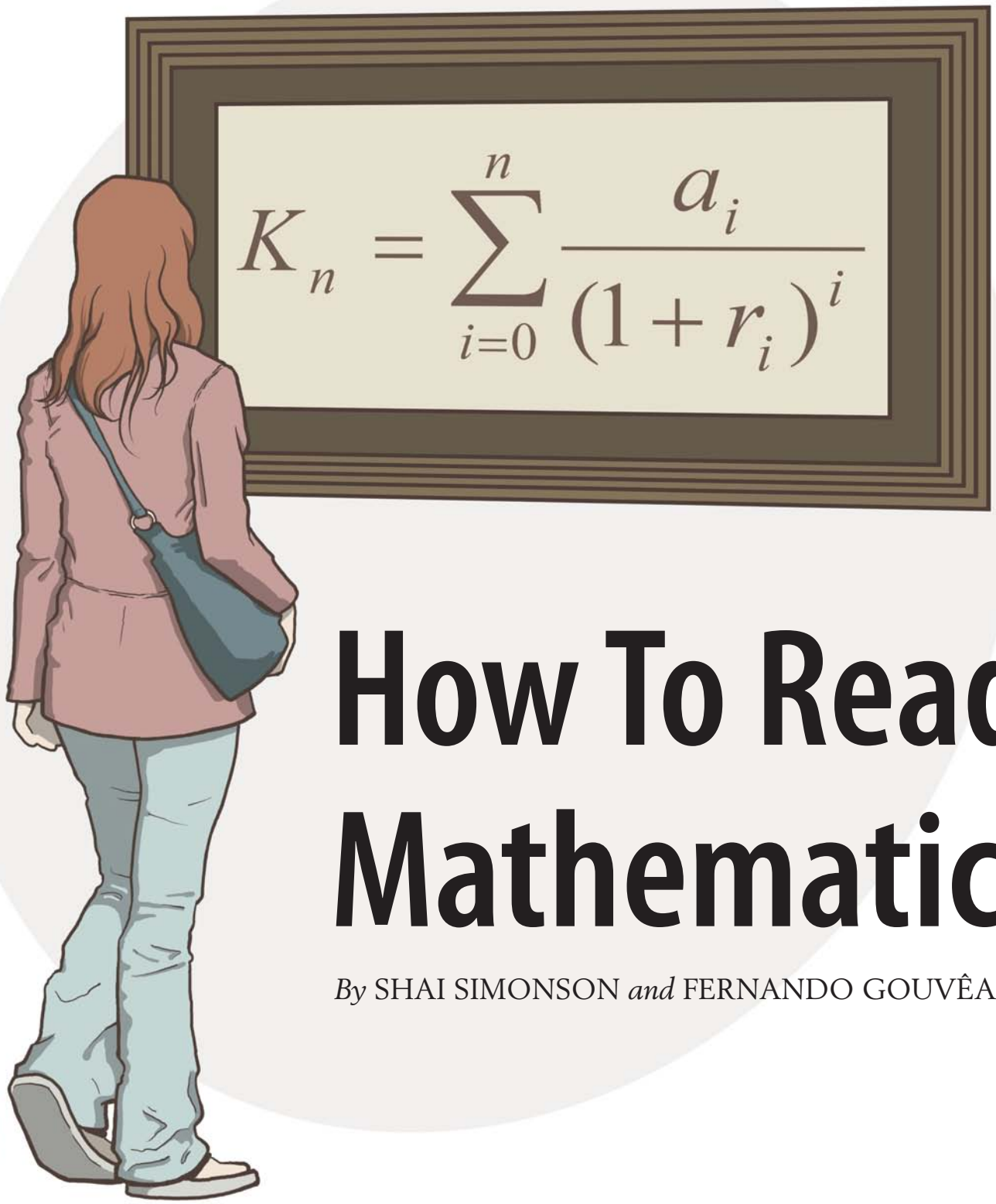


FEATURES

PROGRAMMING

SPECIAL

Illustration: Jaime G. Wong



How To Read Mathematics

By SHAI SIMONSON *and* FERNANDO GOUVÊA

Illustration: Jaime G. Wong

A reading protocol is a set of strategies that a reader must use in order to benefit fully from reading the text. Poetry calls for a different set of strategies than fiction, and fiction a different set than non-fiction. It would be ridiculous to read fiction and ask oneself what is the author's source for the assertion that the hero is blond and tanned; it would be wrong to read non-fiction and not ask such a question. This reading protocol extends to a *viewing* or *listening* protocol in art and music. Indeed, much of the introductory course material in literature, music, and art is spent teaching these protocols.

Ed Rothstein's book, *Emblems of Mind*, a fascinating book that focuses on the relationship between mathematics and music, touches implicitly on reading protocols for mathematics.

> *[Mathematics is] "a language that can neither be read nor understood without initiation."*
> – Emblems of Mind, Edward Rothstein, Harper Perennial, 1996, page 15.

Mathematics has a reading protocol all its own, and just as we learn how to read a novel or a poem, listen to music, or view a painting, we should learn to read mathematics. When we read a novel we become absorbed in the plot and characters. We try to follow the various plot lines and how each affects the development of the characters. We make sure that the characters become real people to us, both those we admire and those we despise. We do not stop at every word, but imagine the words as brushstrokes in a painting. Even if we are not familiar with a particular word, we can still see the whole picture. We rarely stop to think about individual phrases and sentences. Instead, we let the novel sweep us along with its flow, and carry us swiftly to the end. The experience is rewarding, relaxing, and thought provoking.

Novelists frequently describe characters by involving them in well-chosen anecdotes, rather than by describing them by well-chosen adjectives. They portray one aspect, then another, then the first again in a new light and so on, as the whole picture grows and comes more and more into focus. This is the way to communicate complex thoughts that defy precise definition.

Mathematical ideas are by nature precise and well defined, so that a precise description is possible in a very short space. Both a mathematics article and a novel are telling a story and developing complex ideas, but a math article does the job with a tiny fraction of the words and symbols of those used in a novel. The beauty in a novel is in the aesthetic way it uses language to evoke emotions and present themes which defy precise definition. The beauty in a mathematics article is in the elegant way it concisely describes precise ideas of great complexity.

What are the common mistakes people make in trying to read mathematics, and how can these mistakes be corrected?

Don't Miss the Big Picture

> *"Reading Mathematics is not at all a linear experience …Understanding the text requires cross references, scanning, pausing and revisiting"*
> – Emblems of Mind, page 16.

Don't assume that understanding each phrase, will enable you to understand the whole idea. This is like trying to see a portrait by staring at each square inch of it from the distance of your nose. You will see the detail, texture and color but miss the portrait completely. A math article tells a story. Try to see what the story is before you delve into the details. You can go in for a closer look once you have built a framework of understanding. Do this just as you might reread a novel.

Don't be a Passive Reader

> *"A three-line proof of a subtle theorem is the distillation of years of activity. Reading mathematics… involves a return to the thinking that went into the writing"*
> – ibid, page 38.

Explore examples for patterns. Try special cases.

A math article usually tells only a small piece of a much larger and longer story. The author usually spends months exploring things and going down blind alleys. After a period of exploration, experiment, and discovery, the author organizes his/her conclusions into a story that covers up all the mistakes, wrong turns, and associated motivation, presenting the completed idea in a neat linear flow. The way to deeply understand the author's idea is to recreate what the author left out.

There is a lot between the lines of a polished mathematical exposition. The reader must participate. At every stage, he/she must decide whether or not the idea being presented is clear. Ask yourself these questions:

- Why is this idea true?
- Do I really believe it?
- Could I convince someone else that it is true?
- Why didn't the author use a different argument?
- Do I have a better argument or method of explaining the idea?
- Why didn't the author explain it the way that I understand it?
- Is my way wrong?
- Do I really get the idea?
- Am I missing some subtlety?
- Did the author miss a subtlety?
- If I can't understand the idea, can I understand a similar but simpler concept?
- If so, which simpler concept?

- Is it really necessary to understand this idea?
- Can I accept this point without understanding the details of why it is true?
- Will my understanding of the whole story suffer from not understanding why the point is true?

Putting too little effort into this participation is like reading a novel without concentrating. After half an hour, you realize the pages have turned, but you have been daydreaming and don't remember a thing you read.

Don't Read Too Fast

Reading mathematics too quickly results in frustration. When reading a novel, a half hour of concentration might net the average reader 20-60 pages with full comprehension, depending on the novel and the experience of the reader. The same half hour buys you 0-10 lines of a math article, depending on the article and how experienced you are at reading mathematics.

For example, consider the following theorem from Levi ben Gershon's book, Maaseh Hoshev (The Art of Calculation), written in 1321.

> *"When you add consecutive numbers starting with one, and the number of numbers you add is odd, the result is equal to the product of the middle number among them times the last number."*

It is natural for modern day mathematicians to write this as:

$$\sum_{i=1}^{2k+1} i=(k+1)(2k+1)$$

A reader should take as much time to unravel the two-inch version as he would to unravel the two-line version.

Challenge: What does the expression $\sum_{i=1}^{2k+1} i$ mean?

Solution: Given an integer k, add up the values of i, as i ranges from 1 through $2k+1$. In other words, $1 + 2 + 3 + \ldots + 2k+1$.

Challenge: Explain the meaning of $2k+1$ in the expression $\sum_{i=1}^{2k+1} i$

Solution: It represents an odd number. Every even number is two times something, so every odd number is two times something plus one.

Challenge: On the right side of the equation $\sum_{i=1}^{2k+1} i=(k+1)(2k+1)$ the term $2k+1$ represents the last number in the sum. What does $k+1$ represent and why?

Solution: The expression $k+1$ represents the middle number in the sum. The middle number is halfway between 1 and $2k+1$. That is, the middle number equals

$$(1 + 2k+1)/2 = (2k+2)/2 = k+1.$$

Challenge: Can you provide an illustration of Levi's theorem?

Solution: An illustration of Levi's theorem is $\sum_{i=1}^{2(2)+1} i = 1 + 2 + 3 + 4 + 5 = 3\times5$. In this case, $k = 2$.

Challenge: Why is this theorem true?

Solution: The following proof is from one of my students. Her idea is to pair up all the numbers except the last, creating k pairs each of which sums to $2k+1$. Start with the middle pair, k and $k+1$. This pair sums to $2k+1$. Continue pairing numbers moving left from k and right from $k+1$. Each new pair also sums to $2k+1$ since moving left subtracts one and moving right adds one. The last pair is 1 and $2k$, giving k pairs all together. The last number, $2k+1$, is left unpaired. The total sum equals the sum of the k pairs plus the last number, $2k+1$. That is, the total sum equals $k(2k+1) + 2k+1 = (k+1)(2k+1)$.

In contrast, here is Levi's elegant proof discussed in Chapter 4. Levi's proof is similar to my student's but he pairs up numbers starting with the pair surrounding the middle term, k and $k+2$, and working outward. He points out that each pair sums to twice the middle term. This continues until the final pair of numbers, 1 and $2k+1$. Therefore, the entire sum is the same as if every one of the $2k+1$ terms were the middle term, $k+1$. That is, the sum is $(k+1)(2k+1)$.

You can speed up your math reading skill by practicing, but there is no shortcut. Like learning any skill, trying too much too fast can set you back, and may kill your motivation. Imagine joining a high-energy aerobics class when you have not worked out for two years. You may make it through the first class, but you are not likely to come back. The frustration from seeing the experienced class members effortlessly do twice as much as you, while you moan the whole next day from soreness, might be too much to take. Be realistic, be patient, and don't punish yourself.

Make the Idea your Own

The best way to understand what you are reading is to make the idea your own. This means following the idea back to its origin, and rediscovering it for yourself. Mathematicians often say that to understand something you must first read it, then write it down in your own words, then teach it to someone else. Everyone has a different set of tools and a different level of "chunking up" complicated ideas. Make the idea fit in with your own perspective and experience.

"When I use a word, it means just what I choose it to mean"

> *"The meaning is rarely completely transparent, because every symbol or word already represents an extraordinary condensation of concept and reference"*
> – Emblems of Mind, page 16.

A well-written mathematical text will be careful to use a word in one sense only, making a distinction, say, between combination and permutation (or arrangement). A strict mathematical definition might imply that "yellow rabid dog" and "rabid yellow dog" are different arrangements of words but the same combination of words. Most English speakers would disagree. This extreme precision is utterly foreign to most fiction and poetry writing, where using multiple words, synonyms, and varying descriptions is de rigueur. A reader is expected to know that an absolute value is not about some value that happens to be absolute, nor is a function about anything functional.

A particular notorious example of a phrase commonly used in mathematical writing that might easily be misinterpreted is the use of "It follows easily that" and equivalent constructs. The phrase means something like this:

> *One can now check that the next statement is true with a certain amount of essentially mechanical, though perhaps laborious, checking. I, the author, could do it, but it would use up a large amount of space and perhaps not accomplish much, since it'd be best for you to go ahead and do the computation to clarify for yourself what's going on here. I promise that no new ideas are involved, though of course you might need to think a little in order to find just the right combination of good ideas to apply.*

In other words, the construct "It follows easily that," when used correctly, is a signal to the reader that what's involved here is perhaps tedious and even difficult, but involves no deep insights. The reader is then free to decide whether the level of understanding he/she desires requires going through the details or instead, warrants saying "Okay, I'll accept your word for it."

Now, regardless of your opinion about whether that construct should be used in a particular situation, or whether authors always use it correctly, you should understand what it is supposed to mean. "It follows easily that" does not mean

if you can't see this at once, you're a dope,

nor does it mean

this shouldn't take more than two minutes,

but a person who doesn't know the lingo might misinterpret the phrase, and thereby feel frustrated. This is apart from the issue that one person's tedious task is another person's challenge, so not only must the audience engage the author, but the author must correctly judge the audience.

Know Thyself

Texts are written with a specific audience in mind. Make sure that you are the intended audience, or be willing to do what it takes to become the intended audience.

For example, take T.S. Eliot's A Song for Simeon:

> *Lord, the Roman hyacinths are blooming in bowls and*
> *The winter sun creeps by the snow hills;*
> *The stubborn season has made stand.*
> *My life is light, waiting for the death wind,*
> *Like a feather on the back of my hand.*
> *Dust in sunlight and memory in corners*
> *Wait for the wind that chills towards the dead land.*

Eliot's poem pretty much assumes that a reader is going to either know who Simeon was or be willing to find out. It also assumes a reader will be somewhat experienced in reading poetry and/or are willing to work to gain such experience. Eliot assumes that a reader will either know or investigate the allusions here. This goes beyond knowledge of things like who Simeon was. For example, why are the hyacinths "Roman?" Why is that important?

Eliot assumes that the reader will read slowly and pay attention to the images: he juxtaposes dust and memory, relates old age to winter, compares waiting for death with a feather on the back of the hand, and so on. He assumes that a reader will recognize this as poetry; in a way, Eliot is assuming that the reader is familiar with a whole poetic tradition. For example, a reader is supposed to notice that alternate lines rhyme, but that the others do not. Most of all, the poet assumes that a reader will read not only with the mind, but also with his/her emotions and imagination, allowing the images to summon up this old man, tired of life but hanging on, waiting expectantly for some crucial event, for something to happen.

Most math books are written with the assumption that the audience knows certain things, that they have a certain level of "mathematical maturity," and so on. Before you start to read, make sure you know what the author expects you to know.

An Example of Mathematical Writing

To allow an opportunity to experiment with the guidelines presented here, I am including a small piece of mathematics often called the birthday paradox. It is a concise mathematical article explaining the problem and solving it.

The Birthday Paradox

A professor offers to bet anyone in a class of 30 random students that there are at least two people in the class with the same birthday (month and day, but not necessarily year). Would you accept the bet? What if there were fewer people in the class?

Let the birthdays of n people be uniformly distributed among 365 days of the year (assume no leap years for simplicity). We prove that the probability that at least two people have the same birthday (month and day) equals:

$$1-\left(\frac{365\times364\times363\times\ldots\times(365-n+1)}{365^n}\right)$$

What is the probability that among 30 students in a room, there are at least two or more with the same birthday? For n = 30, the probability of at least one matching birthday is about 71%. This means that with 30 people in your class, the professor should win the bet 71 times out of 100 in the long run. It turns out that with 23 people, she should win about 50% of the time.

Here is the proof: Let $P(n)$ be the probability in question. Let $Q(n) = 1 - P(n)$ be the probability that no two people have a common birthday. Now calculate $Q(n)$ by dividing the number of n birthdays without any duplicates by the total number of n possible birthdays. Then solve for $P(n)$.

The number of n birthdays without duplicates is:

$$365 \times 364 \times 363 \times \ldots \times (365 - n + 1).$$

This is because there are 365 choices for the first birthday, 364 for the next and so on for n birthdays.

The total number of n birthdays without any restriction is $365n$ because there are 365 choices for each of n birthdays. Therefore, $Q(n)$ equals

$$\frac{365\times364\times363\times\ldots\times(365-n+1)}{365^n}$$

Solving for $P(n)$ gives $P(n) = 1 - Q(n)$ and hence our result. ■

Shai Simonson received his BA in mathematics from Columbia University and his PhD in computer science from Northwestern. Currently, he is a professor at Stonehill College. Simonson has taught mathematics and computer science to students from middle school through graduate school for over 30 years, and recently published a Java textbook with McGraw Hill. The article How to Read Mathematics appears in his newest book Rediscovering Mathematics, (coming out late 2010). The book is recommended for general readers, and contains no math past 10th grade.

Fernando Q. Gouvêa is Carter Professor of Mathematics at Colby College. For the last 11 years, he was editor of MAA Focus, the newsmagazine of the Mathematical Association of America, and is currently editor of MAA Reviews, an online book review service. A specialist in Number Theory and the History of Mathematics, Gouvêa is the author of several books, including Math through the Ages: A Gentle History for Teachers and Others, co-authored with William P. Berlinghoff.

Reprinted with permission of the original authors. First appeared in *http://hn.my/readmath/*.

BREADPIG SEARCHED THE GALAXIES FOR THE AWESOMEST SAUCE

...and found this 12 oz bottle of hot garlicky goodness. Unicorn tested, TIE-fighter Darwin approved.

Breadpig donates all of its non-sustainable profits to organizations and individuals doing great things for the world. We spend the rest of our time discovering and promoting fascinating technology, hacks, and ideas from all over the world that inspire and impress us. Find out more at www.breadpig.com.

GET AWESOMESAUCE AT STORE.XKCD.COM/BREADPIG OR FROM OUR FRIENDS AT THINKGEEK.

Illustration: Pasquale D'Silva

Why Free Plans Don't Work

By RUBEN GAMEZ

Not too long ago it seemed like every product I knew was offering some sort of free plan. The strategy was brilliant: get loads of people using your product and eventually turn them into paying customers. Everywhere I looked there were stories of people making money hand over fist with this approach.

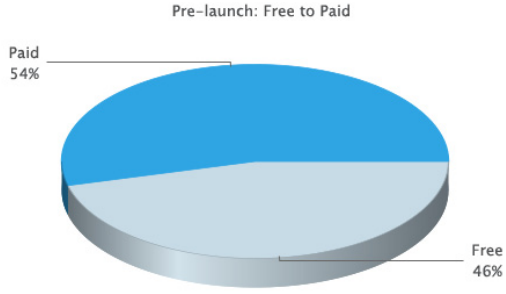
When 37signals talked about giving something away for free as a marketing strategy, it made a lot of sense to me:

> *"For us, Writeboard and Ta-da list are completely free apps that we use to get people on the path to using our other products. Also,* ***we always offer some sort of free version of all our apps****.*
> *We want people to experience the product, the interface, the usefulness of what we've built. Once they're hooked, they're much more likely to upgrade to one of the paying plans (which allow more projects or pages and gives people access to additional features like file uploading and ssl data encryption)."*

So when I launched Bidsketch — a SaaS based proposal application for designers — offering a free plan was a no-brainer in my book. Out of all the important decisions I spent time mulling over before my launch, I gave this one the least thought.

Early on, things were working out nicely. In the first few days of my launch I had more people sign up for the paid plan than the free plan.

"*Man, this free plan is really working out*," I thought. Here is a look at the numbers:



The numbers looked but great, but I suspected they weren't sustainable because I had launched to my mailing list. A well-maintained mailing list tends to convert much better than traffic from other sources.

In any case, I was still happy with the results a week later, once I started converting general website traffic:

While the numbers looked good I knew they wouldn't last because I was relying on a limited time offer. I just didn't realize how much worse things would get:

For the next month only 1% of users would choose the paid option. My user base was growing fast but the money was barely trickling in. Also, support was starting to get tricky, which left me uncomfortable at the thought of what things would look like six months down the line.

> **"If I stayed on this path, I'd soon have thousands of free users to support."**

How many of the free accounts was I able to upgrade to paid? I didn't fare any better upselling users: 0.8% of free user accounts eventually upgraded to paid.

When things started going south, I figured I was to blame for this. I simply wasn't carving out the right features. Or maybe I wasn't prompting for upgrades at the right places. I tried all sorts of tactics to convert my free users:

- More upgrade prompts
- Less features on free accounts
- Premium features for 15 days
- More emails aimed at upselling users to paid

None of these changes had a significant impact. The only thing that seemed to be consistent about my growth was that my revenue was relatively flat while my user base kept growing.

If I stayed on this path, I'd soon have thousands of free users to support.

So in a desperate attempt to get things moving in the right direction, I experimented for a week by killing my free plan. I didn't tell anyone that I was getting rid of my free plan, I simply deleted it from my pricing page.

My major concern was that I'd keep the same number of paid users coming in and I'd lose all the free ones. Which means I wouldn't have a targeted list of users to try to upsell to a paid plan. Not that I was having much success getting them to upgrade, but at least it was something.

Things didn't quite turn out that way. This change that took all of five minutes to make, led to an **8x increase in paid conversions**.

Look at that again. That's not 8%. That's 800%.

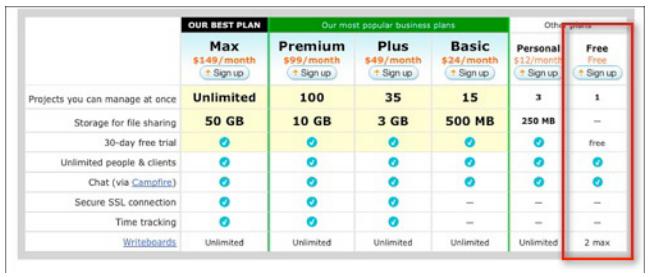
I felt comfortable enough with the results to try it out for the entire month. Amazingly, this resulted in a 10x increase in paid conversions for the month.

And I'm not the only one

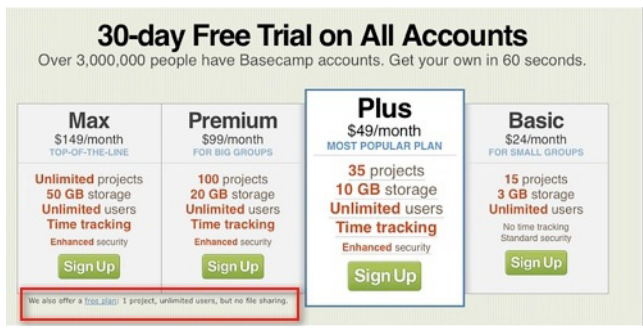
It wasn't long after I got rid of my free plan that I started to notice that a lot of people were citing similar issues with having a free plan.

I saw that 37signals had hidden theirs.

Before:



After:



And that I ran into a Mixergy interview where 37signals founder, Jason Fried, talks about their free plan (6:00 into the interview):

> *"… The majority of the revenues for our products come from people who sign up for the paid versions upfront. So we definitely have people upgrading from free to paid, **but the majority of people who are on pay started on pay**… of course, more people are going to pick the free version and stay on the free version, but if you're looking to get paying customers, ask for money upfront and you'll have a lot better shot of getting them."*

The so-called Freemium success stories had similarly low ratios of free to paid accounts. We can see numbers published about Pandora, Evernote, and MailChimp showing this pattern.

Pandora started out with less than 1% of their user base as paid subscribers. Once they focused on delivering a better premium offering they were able to increase that to 1.7%. Still, pretty underwhelming unless you've got 20 million people using your service like they do.

> **"Taking advantage of word-of-mouth marketing requires more users than most of us will attain."**

Evernote is looking at a 0.5% conversion rate to paid accounts initially and can convert 2% of the people that stick around for a year.

While there wasn't a specific conversion rate published for MailChimp, they did mention the negative side effect of abuse-related issues:

> *"But the biggest bumps of all? A 354 percent increase in abuse-related issues like spamming, followed by a 245 percent increase in legal costs dealing people trying to game the system."*

Holy crap. Where was this info when I needed it?

CrazyEgg decided to drop its free plan in Jan of 2009 and they haven't looked back. I asked CrazyEgg co-founder Hiten Shah why they decided to drop their free plan back then. "We thought that if we dropped it we would make more money," said Hiten. This turned out to be a good move since it doubled their revenue that month.

LessAccounting co-founder Allan Branch said while they don't claim to know what the best approach is in regards to a free plan, they haven't seen a good reason to change what they're doing now. With them, users have to sign up to a paid plan trial, and will get dropped to a free plan if they don't enter payment information at the end of the trial. Obviously, this approach of making users choose a paid plan at signup has worked well for them so far.

An Example We Can Relate To

A lot of us aren't at the same level that these guys are; we're not dealing with millions of users, or even hundreds of thousands. So an example like Pluggio might be easier to relate to.

Pluggio is a Freemium Twitter web app created by Justin Vincent. He has a great stats page that shows everything from monthly revenue to the breakdown of users by plan type.

Taking a look at that page reveals that he's actually doing very well for a relatively young app in this space.

He's been averaging about a thousand dollars a month since November of last year. And unlike the bigger guys, his paid users make up 2.5% of all accounts. That's damn good for any sort of Freemium app judging by the numbers that we've seen so far.

I spoke with Justin to ask him about his experience with the Freemium model. He seemed to be doing well with Pluggio which is why I was surprised when he told me he was seriously considering killing his free plan.

His reason for doing this? Revenue has been relatively flat and the number of users has been steadily increasing over the last few months (currently nearing five thousand).

This says a lot about the pitfalls of having a free plan for entrepreneurs with limited resources.

Do they ever make sense?

I'm not saying that it's impossible to be successful if you launch with a free plan.

Obviously free plans have worked well for companies like Wufoo, MailChimp, and FreshBooks, so we know they can work. But the problem is that **we're not them**.

We need to stop blindly copying them and start thinking about ways to bring in revenue.

I'll concede that there are certain types of apps that are more likely to succeed by offering a free plan and going with the Freemium model. But the vast majority of apps aren't in this category, and the vast majority of people don't have the resources to make that model work.

Taking advantage of word-of-mouth marketing requires more users than most of us will attain. Instead, we end up with a large number of free users zapping away valuable resources for nothing in return. To top it off, most free users will never end up converting to a paid plan.

If we have thousands of users that don't increase awareness and will never pay for our product, why do we insist in offering something that's going to hurt our business? Maybe we should just skip that free plan and focus on making money instead. ■

Ruben Gamez is the founder of Bidsketch, web based proposal software for designers. When he's not developing software, he's furiously working towards becoming a better Micropreneur.

Reprinted with permission of the original author. First appeared in *http://hn.my/free/*.

Rich Programmer Food

By STEVE YEGGE

This is another one of those blog topics I've been sitting on for way too long, trying to find a polite way of saying something fundamentally impolite. I don't see a way to do it. So: you stand a good chance of being offended by this blog entry. (Hey, just don't say I didn't warn ya.)

I've turned off blog comments, incidentally, because clever evil people have figured out how to beat captchas using non-algorithmic approaches, and I don't have the bandwidth to police spam myself. Sorry.

I don't want to give you a heart attack, so I'm going to give you the gentle-yet-insistent executive summary right up front. If you can make it through my executive summary without a significant increase in heart rate, then you're probably OK. Otherwise, you might consider drinking heavily before reading this, just like people did in the old movies when they needed their leg sawed off. That's what I'm doing, in any case (drinking, that is, not sawing my leg off).

Gentle, yet insistent executive summary: If you don't know how compilers work, then you don't know how computers work. If you're not 100% sure whether you know how compilers work, then you don't know how they work.

You have to *know* you know, you know.

In fact, Compiler Construction is, in my own humble and probably embarrassingly wrong opinion, the second most important CS class you can take in an undergraduate computer science program.

Because every deep-dive I've attempted on this topic over the past year or so has failed utterly at convincing me after I sobered up, I'm going to stage this production as a, erm, stage production, with N glorious, er, parts, separated by intermissions. So without further ado...

Actually, that sounds like way too much work. So I'll just rant. That's what you paid good money to hear anyway, right? I promise to make so much fun of other people that when I make fun of you, you'll hardly notice.

Cots and Beards

I took compilers in school. Yup. Sure did. From Professor David Notkin at the University of Washington, circa late 1991 or thereabouts.

Guess what grade I got? I got a zero. As in, 0.0. That was my final grade, on my transcript. That's what happens at the University of Washington when you get an Incomplete and don't take the necessary corrective actions (which I've never figured out, by the way.) After some time elapses, it turns into a zero.

You can get an Incomplete in various different legitimate ways, including my way, which was to be an ill-considered beef-witted mooncalf who takes the course past the drop-date and then decides not to finish it because he doesn't feel like it. I *earned* that Incomplete, I tell you.

I took Compilers again a few years later. I was in college for a long time, because I got hired on as a full-time employee by Geoworks about a year before I graduated (among other reasons), and it wound up extending my graduation for several years.

Don't do that, by the way. It's really hard to finish when you're working full-time. Get your degree, *then* go to work. All the more so if you're a Ph.D. candidate within any reach of finishing. You don't want to be just another ABD for the rest of your life. Even if you're not sad, per se, we'll be sad for you.

I got a decent grade in Compilers the second time around. I actually understood compilers at a reasonably superficial level the first time, and not too badly the second time. What I failed to grasp for many more years, and I'm telling you this to save you that pain, is *why* compilers actually matter in the first place.

> **“The Olive Garden: it’s where poor people go to eat rich people food.”**
>
> — Dave Yegge

Here’s what I thought when I took it back in 1991. See if it sounds familiar. I thought: a compiler is a tool that takes *my* program, after whining about it a lot, and turns it into computer-speak. If you want to write programs, then a compiler is just one of those things you need. You need a computer, a keyboard, an account maybe, a compiler, an editor, optionally a debugger, and you’re good to go. You know how to Program. You’re a Programmer. Now you just need to learn APIs and stuff.

Whenever I gave even a moment’s thought to whether I needed to learn compilers, I’d think: I would need to know how compilers work in one of two scenarios. The first scenario is that I go work at Microsoft and somehow wind up in the Visual C++ group. Then I’d need to know how compilers work. The second scenario is that the urge suddenly comes upon me to grow a long beard and stop showering and make a pilgrimage to MIT where I beg Richard Stallman to let me live in a cot in some hallway and work on GCC with him like some sort of Jesuit vagabond.

Both scenarios seemed pretty unlikely to me at the time, although if push came to shove, a cot and beard didn’t seem all that bad compared to working at Microsoft.

By the way, my brother Dave was at a party once, long ago, that had more than its fair share of Microsoft people, and apparently there was some windbag there bragging loudly (this is a *party*, mind you) that he had 15 of the world’s best compiler writers working for him in the Visual C++ group. I told Dave: “wow, I didn’t realize Richard Stallman worked at Microsoft”, and Dave was bummed that he hadn’t thought of that particular riposte at the time. So it goes.

The sad part about that story is that I’ve found myself occasionally wanting to brag that I work with some of the best compiler writers in the world at Google. Please, I beg you: if you ever find me at a party bragging about the compiler writers I work with, have pity on us all and shoot me dead on the spot. Hell, bash me over the head with a lamp if you have to.

Anyway, now you know what I thought of compilers in 1991. Why I even took the class is beyond me. But I didn’t finish. And the second time around – which I only did because I felt bad about the first time around: not from the zero, but from having let David Notkin down – I only took the time to understand the material well enough to finish the course with a decent grade.

I was by no means atypical. If you’re a CS student and you love compilers (which, anecdotally, often means you’re in the top 5% of computer science students in your class worldwide), then I salute you. I bet I’m way better at Nethack than you are. The reality is that most programmers are just like I was, and I can’t really fault ’em for that.

Before I leave this sordid story forever, I feel obliged to point out that it’s *partly* academia’s fault. Except for type systems research, which is being treated with approximately the same scholarly caution and restraint as Arthur’s Grail Quest, compilers have been out of favor in academia for a long time. So schools don’t do a good job at marketing compilers, and giving them due credit as a critical topic in their own right. It’s a sad fact that most schools don’t require you to take compilers in order to graduate with a Computer Science degree.

Sigh.

How Would You Solve...

You're a programmer, right? OK, I'll propose some programming situations for you, and you tell me how you'd solve them.

Situation ❶: you're doing a bunch of Java programming, and your company has explicit and non-negotiable guidelines as to how to format your Java code, down to every last imaginable detail. How do you configure your editor to auto-format your code according to the style guide?

Situation ❷: your company does a lot of Ajax stuff, and your JavaScript code base is growing almost as fast as your other code. You decide to start using jsdoc, a javadoc pseudoclone for JavaScript, to document your functions in a way that permits automated doc extraction. You discover that jsdoc is a miserable sod of a Perl script that seg faults on about 50% of your code base, and – bear with me here – you've vowed never to write another line of Perl, because, well, it's Perl. Pick your favorite reason. How do you write your own jsdoc extractor, bearing in mind that it will need to do at least a cursory parse of the JavaScript code itself?

Situation ❸: your company has a massive C++ code base, the result of many years of hard work by dozens, if not hundreds, of engineers. You discover that the code needs to be refactored in a nontrivial way, e.g. to upgrade from 32-bit to 64-bit, or to change the way you do your database transactions, or (God help you) because you're upgrading your C++ compiler and the syntax and semantics have all changed again. You're tasked with fixing it. What do you do?

Situation ❹: someone at your company writes a bitchin' new web based code review tool. Everyone switches to it. You realize, after using it for a while, that you miss having it syntax-highlight the source code for you. You don't have much time, but you might be able to afford a week or so, part-time, to make it happen. How do you do it? (Let's say your company uses five to eight languages for 99% of their code.)

Situation ❺: an unexpected and slightly bizarre new requirement arises on your current project: you need to be able to use a new kind of hardware router. Maybe all your Web 2.0 stuff is screwing up your border routers or network bandwidth monitors, who knows. All you know is the sysops and network engineers are telling you that you need to talk to these new routers directly. The routers have IP addresses, a telnet interface, and a proprietary command language. You send commands, and they send responses. Each command has its own syntax for its arguments, and you need to parse the responses (which have no documented format, but you can reverse-engineer it) to look for certain patterns, in order to set them in the right state for your wacky uploads or downloads. What tool do you use?

Situation ❻: your company's projects are starting to slip. The engineers are all smart, and they are all using the latest and greatest state-of-the-art Agile Object-Oriented Software Engineering Principles and programming languages. They are utterly blameless. However, for some reason your code base is getting so complex that project estimates are going wildly awry. Simple tasks seem to take forever. The engineers begin talking about a redesign. This is the Nth such redesign they have gone through in the past five years, but this is going to be the big one that fixes everything. What color slips of paper do you give them? Woah, ahem, sorry, I mean how do you ensure their success this time around?

Situation ❼: you have a small, lightweight startup company filled with cool young people with long blue-tinted hair and nose rings and tongue rivets and hip black clothes and iphones and whatever the hell else young people have these days. You use Ruby on Rails for your site, and it scales just fine for your number of visitors. (You've never bothered to measure whether your number of visitors is a function of your site's latency, because it's never occurred to you to wonder.) You read about the latest horrible godawful Rails security vulnerability, under which users can make arbitrary SEC filings on behalf of your company by sending properly formatted GET requests to your public site. You download the new version and read the unit test code to figure out what the actual vulnerability is, since they didn't say, and you determine that you need to make a set of nontrivial code changes to remove a particular (and mysteriously non-greppable) idiom from your code base, replacing it by mechanical transformation to a different idiom. How do you do it?

Situation ❽: some drunken blogger presents you with seven weird situations and asks you to speculate about what they have in common. Do you already know the answer?

Here are the answers. What, you thought these were rhetorical?

Scenario ❶: you lobby your company to change the style guide to match whatever Eclipse does by default.
Scenario ❷: you post to the jsdoc mailing list and ask if anyone else has had this problem. Several people say they have, and the issue pretty much dies right then and there.
Scenario ❸: You quit. Duh. You knew that was the answer before you reached the first comma.
Scenario ❹: Tough it out. Colors are for weenies. Or maybe you wire up GNU Source Highlight, which covers languages all the way from Fortran to Ada, and you live with the broken highlighting it provides.
Scenario ❺: Perl. It's a swiss army knife. You can use it to sidestep this problem with honor, by disemboweling yourself.
Scenario ❻: Pink.
Scenario ❼: Fix it by hand. Hell, you only have about 10k lines of code for your whole site. It's Rails, fer cryin' out loud. This was a trick question.
Scenario ❽: Yes. You skim until the end of the blog, just to find out what the first-most-important CS class is. Stevey's well known for shaggy-dog jokes like this.

And there you have it. You're now equipped to deal with just about every programming situation you could come across. So you obviously don't need to know compilers.

How Compilers Work

Here are some real-life answers from real-life candidates, with real-life Ph.D.s in Computer Science, when asked how compilers work.

Real Candidate ①: "Oh! They, ah, um, scan your program one line at a time and convert each line to assembly language."
Real Candidate ②: "Compilers check errors in your program and, ah, tell you if you had bad syntax. That's all I remember."
Real Candidate ③: "I... <3-minute pause>... I don't know."
Real Candidate ④: "They preprocess your program and convert #DEFINE statements into code, and then, um, emit machine code."

That's pretty much all the detail you'll ever get out of 75% of all interview candidates, because, hey, they don't want to work in a hallway at MIT. Can you blame them?

Only about 3% to 5% of all interview candidates (and that's being optimistic) can tell you any details about how a compiler works. The rest will do some handwaving about lex and yacc and code generation, maybe.

I told you your heart rate would go up. Didn't I?

Take a deep breath.

Why Compilers Matter, Part 1

The first reason Compiler Construction is such an important CS course is that it brings together, in a very concrete way, almost everything you learned before you took the course.

You can't fully understand how compilers work without knowing machine architecture, because compilers emit machine code. It's more than just instructions; compilers need to understand how the underlying machine actually operates in order to translate your source code efficiently.

Incidentally, "machines" are just about anything that can do computations. Perl is a machine. Your OS is a machine. Emacs is a machine. If you could prove your washing machine is Turing complete, then you could write a compiler that executes C code on it.

But you knew that already.

You can't understand how modern compilers work without knowing how Operating Systems work, because no self-respecting machine these days runs without an operating system. The OS interface forms part of the target machine. Sure, you can find people working on five- to ten-year mainframe projects that ultimately run no faster than a PC from Costco, and they may dispense with the operating system due to time constraints, plus the fact that they have a worldwide market of one customer. But for most of us, the OS is part of the machine.

You won't understand how compilers work unless you've taken a theory of computation course. The theory of computation reads like part one of chapter 1 of a compilers book. You need all of it.

You'll have difficulty keeping the phases (and even the inputs and outputs) of a compiler straight in your head unless you've taken a programming languages course. You have to know what the capabilities of programming languages are, or at least have an inkling, before you can write a program that implements them. And unless you know more than one language well, it won't make much sense to write a program in language A that converts language B to language C.

You're actually surrounded by compilation problems. You run into them almost every day. The seven scenarios I outlined above are the tip of the iceberg. (The eighth one is the rest of the iceberg, but no skimming!)

Compilers take a stream of symbols, figure out their structure according to some domain-specific predefined rules, and transform them into another symbol stream.

Sounds pretty general, doesn't it? Well, yeah.

Could an image be considered a symbol stream? Sure. Stream each row of pixels. Each pixel is a number. A number is a symbol. You can transform images with compilers.

Could English be considered a symbol stream? Sure. The rules are pretty damn complex, but yes, natural language processing (at least the deterministic kind that doesn't work and has been supplanted by stochastic methods) can be considered a fancy kind of compilation.

What about ordinary code? I mean, we don't all deal with image processing, or natural language processing. What about the rest of us? We just write code, so do compilers really matter?

Well, do you ever, EVER need to write code that deals with your own code base? What if you need to write a syntax highlighter? What if your programming language adds some new features and your editor doesn't support them yet? Do you just sit around and wait for "someone" to fix your editor? What if it takes years? Doesn't it seem like you, as the perfectly good programmer that you are, ought to be able to fix it faster than that?

Do you ever need to process your code base looking for certain idioms? Do you ever need to write your own doc extractor?

Have you ever worked on code bases that have grown inexplicably huge, despite all your best efforts to make them modular and object-oriented? Of course you have. What's the solution?

You either learn compilers and start writing your own DSLs, or your get yourself a better language.

I recommend NBL, by the way. It's my personal favorite: the local maximum in a tensor-field of evil, the highest ground in the vicinity of Hell itself. I'm not going to tell you what NBL is, yet, though. Patience! I'm only half done with my Emacs-mode for it.

If you don't take compilers...

One reason many programmers don't take compilers is that they've heard it's really, really hard. It's often the "capstone" course of a CS program (OS often being the other one), which means it's a sort of "optional rite of passage" that makes you a Real Programmer and puts hair on your chest, regardless of gender or chest-hair preference.

If you're trying to plan out a schedule that gets you to graduation before the money runs out, and hopefully with a GPA that doesn't cause prospective employers to summon the guard dogs on you, then when you hear the phrase "optional rite of passage", who can blame you if you look for alternatives?

I'm not saying other CS courses aren't important, incidentally. Operating Systems, Machine Learning, Distributed Computing and Algorithm Design are all arguably just as important as Compiler Construction. Except that you can take them all and still not know how computers work, which to me means that Compilers really needs to be a mandatory 300-level course. But it has so many prerequisites that you can't realistically make that happen at most schools.

Designing an effective undergrad CS degree is hard. It's no wonder so many ivy-league schools have more or less given up and turned into Java Certification shops.

If you're a conscientious CS student, you'll at least take OS and AI. You may come out without knowing exactly how compilers work, which is unfortunate, but there will be many problem domains in which you can deliver at least as much value as all the other people just like you. That's something to feel good about, or at least as good as everyone else feels at any rate.

Go team.

Most programmers these days, sadly, just want the degree. They don't care what they learn. They want a degree so they can get a job so they can pay the bills.

Most programmers gravitate towards a set of courses that can best be described as the olive-garden of computer science: the places where dumb programmers go to learn smart programmer stuff.

I hesitate to name these courses explicitly. I wouldn't be agile enough to dodge the game of graphic bloodshed aimed at me by animated, project-managing, object-oriented engineers using Java and Web 2.0 technologies to roast me via user interfaces designed rationally through teamwork and modern software methodologies. I'd become a case study in the ethics of software and its impact on our culture.

But you can probably imagine what some of the courses are.

If you don't take compilers then you run the risk of forever being on the programmer B-list: the kind of eager young architect who becomes a saturnine old architect who spends a career building large systems and being damned proud of it.

Large Systems Suck

This rule is 100% transitive. If you build one, you suck.

Compiler Camps

It turns out that many compiler "experts" don't know compilers all that well, because compilers can logically be thought of as three separate phases – so separate, in fact, that they constitute entirely different and mostly non-overlapping research domains.

The first big phase of the compilation pipeline is parsing. You need to take your input and turn it into a tree. So you go through preprocessing, lexical analysis (aka tokenization), and then syntax analysis and IR generation. Lexical analysis is usually done with regexps. Syntax analysis is usually done with grammars. You can use recursive descent (most common), or a parser generator (common for smaller languages), or with fancier algorithms that are correspondingly slower to execute. But the output of this pipeline stage is usually a parse tree of some sort.

You can get a hell of a lot farther as a professional programmer just by knowing that much. Even if you have no *idea* how the rest of the compilation works, you can make practical use of tools or algorithms that produce a parse tree. In fact, parsing alone can help you solve Situations #1 through #4 above.

If you don't know how parsing works, you'll do it badly with regular expressions, or if you don't know those, then with hand-rolled state machines that are thousands of lines of incomprehensible code that doesn't actually work.

Really.

In fact I used to ask candidates, as a standard interview question, how they'd find phone numbers in a tree of HTML files, and many of them (up to 30%) chose to write 2500-line C++ programs as their answer.

At some point, candidates started telling me they'd read that one in my blog, which was pretty weird, all things considered. Now I don't ask it anymore.

I ask variants of it occasionally, and it still gets them: you either recognize it as an easy problem or you get out the swiss army knife and start looking for a second to behead you before the pain causes you to dishonor your family.

C++ does that surprisingly often.

The next big phase is Type Checking. This is a group of zealous academics (and their groupies and/or grad students) who believe that they can write programs that are smart enough to figure out what *your* program is trying to do, and tell you when you're wrong. They don't think of themselves as AI people, though, oddly enough, because AI has (wisely) moved beyond deterministic approaches.

This camp has figured out more or less the practical limit of what they can check deterministically, and they have declared that this is the boundary of computation itself, beyond the borders of which you are crossing the outskirts of civilization into kill-or-be-killed territory, also occasionally known as The Awful Place Where People Make Money With Software.

You should hear them when they're drunk at rave parties.

A good friend of mine with a Ph.D. in languages told me recently that it's "very painful" to experience the realization that all those years of slaving over beautiful mathematical purity have more or less zero bearing on the real world.

The problem – well, one problem – is the underlying premise, which is apparently that without the Hindley-Milner type system, or failing that, some crap-ass type system like Java's, you will never be able to write working code; it'll collapse under its own weight: a vast, typeless trap for the unwary adventurer.

They don't get out much, apparently.

Another problem is that they believe any type "error", no matter how insignificant it might be to the operation of your personal program at this particular moment, should be treated as a news item worthy of the Wall Street Journal front page. Everyone should throw down their ploughshares and stop working until it's fixed. The concept of a type "warning" never enters the discussion.

Remember when fuzzy logic came along? Oh, oh, wait — remember when von Neumann and Stan Ulam introduced the Monte Carlo method? Oh, right, I keep forgetting: you were born in nineteen-ninety something, and you're nineteen, and I'm ninety-something.

Well, someday they will realize that strict determinism has always, always failed, in every dimensionality-cursed domain to which it's ever been applied, and it's always replaced by probablistic methods.

Call it "optional static types", as an embryonic version of the glorious future. NBL, anyone?

The third camp, who tends to be the most isolated, is the code generation camp. Code generation is pretty straightforward, assuming you know enough recursion to realize your grandparents weren't Adam and Eve. So I'm really talking about Optimization, which is the art of generating code that is just barely correct enough that most of your customers won't notice problems. Wait, sorry, that's Amazonization. Optimization is the art of producing *correct* code that is equivalent to the naive, expensive code written by your presumably naive, expensive programmers.

I'd call compiler optimization an endless chasm of eternal darkness, except that it's pretty fun. So it's an endless chasm of fun eternal darkness, I guess. But you can take it to extremes you'd never guess were possible, and it's a fertile, open research field, and when they "finish", they'll be in the same place the Type Checking camp wants to be, namely AI experts.

By which I mean Machine Learning, since the term "AI" smacks of not just determinism, but also a distinct lack of VC funding.

In any case, the three camps don't really mingle much, and all of them have a valid claim at calling themselves "compiler experts" at raves.

The Dark Side of Compilers

One of the reasons it took me so long to write this ridiculous blog entry is that I wanted to go write a compiler for myself before I spouted off about them.

Done!

Well, sort of. Actually, "not done" would be more accurate, since that, as I've found, is the steady state for compilers everywhere.

Without giving any details away, as that would be premature, I took a stab at writing an interpreter for a useful language, using another useful language, with the output being useful bytecode for a useful platform.

It was fun. It went pretty fast. I learned a lot, even though I'd taken compilers twice in school 15 years ago, and even though I've gradually taught myself about compilers and programming languages over the past 5 years or so.

I still learned a lot just by doing it.

Unfortunately, writing a compiler creates a living thing. I didn't realize this going into it. I wasn't asking for a baby. It was a complete surprise to me, after 20-odd years of industry experience, that even writing a simple interpreter would produce a lifetime of work.

Go figure.

I credit the phrase "a lifetime of work" to Bob Jervis, a friend of mine who happens to be the original author of Turbo C (with which I myself learned to program), and a damn good, even world-class compiler writer.

He gave a tech talk recently (Google does that a LOT) in which he pointed out that even just the set of features the audience had asked for was a lifetime of work.

This phrasing resonated deeply with me. It was similar to my realization about 18 months back that I only have a small finite number of 5-year projects left, and I have to start choosing them very carefully. After writing my own "production interpreter", I realized that the work remaining was unbounded.

I mean it. Unbounded.

So from one perspective, I suppose I should just release what I've got and start marketing it, so other people will jump on board and start helping out. On the other hand, I started this particular side-project not to create a lifetime of work for myself (far from it), but to make sure I knew enough about compilers to be able to rant semi-intelligently about them, after a few glasses of wine, to a quarter million readers.

So I'd at least better finish the byte compiler first.

I'll get there. It'll be neat. I've only described this crazy little side project to a handful of people, and they reacted pretty uniformly by yelling "WTF????" You know, the kind of shout you'd yell out if you discovered the most sane person you knew in the entire world trying to stuff a lit stick of dynamite into their mouth.

That's compilers for ya. You can hardly attempt one without trying to change the world in the process.

That's why you need to learn how they work. That's why you, yes you personally, need to write one.

It's not as hard as you think, except for the fact that it will turn into a lifetime of work. It's OK. You can walk away from it, if you want to. You probably won't want to. You may be forced to, due to time constraints, but you'll still be a far better programmer for the effort.

You'll be able to fix that dang syntax highlighting.

You'll be able to write that doc extractor.

You'll be able to fix the broken indentation in Eclipse.

You won't have to wait for your tools to catch up.

You might even stop bragging about how smart your tools are, how amazing it is that they can understand your code — which, if I may say so, isn't something I'd go broadcasting quite so loudly, but maybe it's just me.

You'll be able to jump in and help fix all those problems with your favorite language. Don't even try to tell me your favorite language doesn't have problems.

You'll be able to vote with confidence against the tired majority when some of the smartest people in the world (like, oh, say, James Gosling and Guy Steele) try to introduce non-broken closures and real extensibility to the Java community. Those poor Java Community schmucks. I pity them all. Really I do.

Heck, you might even start eating rich programmer food. Writing compilers is only the beginning; I never claimed it was the end of the road. You'll finally be able to move past your little service APIs and JavaScript widgets, and start helping to write the program that cures cancer, or all viruses worldwide, or old age and dying. Or even (I'm really going out on a limb here) the delusion of Static Typing as a deterministic panacea.

If nothing else, you'll finally really learn whatever programming language you're writing a compiler for. There's no other way. Sorry!

And with that, I suppose I should wrap up. I'm heading to Foo Camp in the morning, and I have no idea what to expect, but I have a pretty good guess that there won't be much discussion of compilers, except hopefully from GVR vis a vis Python 3000. That might be cool.

If you don't know compilers, don't sweat it. I still think you're a good programmer. But it's good to have stretch goals!

But What's The Most Important CS Course?

Typing 101. Duh.

Hie thee hence. ■

Steve Yegge is a Staff Software Engineer at Google. Prior to Google he worked at *Amazon.com* as a Senior Software Development Manager. He earned his Computer Science degree from the University of Washington, and has over twenty years of experience as a software developer, dev manager, programmer hobbyist and tech blogger. Steve's current interests include Clojure, GNU Emacs and GNU Lilypond.

Reprinted with permission of the original author. First appeared in *http://hn.my/compiler/*.

We're not going to lie.
Dropbox is a pretty sweet place to work.

Dropbox is revolutionizing the way people think about their files. Since our founding in 2008 by Drew and Arash, we've attracted millions of users, and we sync petabytes of their data across nearly every platform imaginable.

With a small team of engineers, we've created a sick product, and we're looking for more people to join our team to continue this tradition.

We offer competitive salaries, a handful of benefits, and a great environment in San Francisco. You'll own projects that address challenging problems, and work with people who know where you're coming from.

At Dropbox, you can start a project on Monday and have it seen by millions on Friday. Afterward, chill by jamming with the team band or by playing some in-house SC2.

Sound good? Head here: http://dropbox.com/jobs

Readme Driven Development

By TOM PRESTON-WERNER

I HEAR A LOT of talk these days about TDD and BDD and Extreme Programming and SCRUM and stand up meetings and all kinds of methodologies and techniques for developing better software, but it's all irrelevant unless the software we're building meets the needs of those that are using it. Let me put that another way. A perfect implementation of the wrong specification is worthless. By the same principle a beautifully crafted library with no documentation is also damn near worthless. If your software solves the wrong problem or nobody can figure out how to use it, there's something very bad going on.

Fine. So how do we solve this problem? It's easier than you think, and it's important enough to warrant its very own paragraph.

Write your Readme first.

First. As in, before you write any code or tests or behaviors or stories or ANYTHING. I know, I know, we're programmers, dammit, not tech writers! But that's where you're wrong. Writing a Readme is absolutely essential to writing good software. Until you've written about your software, you have no idea what you'll be coding. Between The Great Backlash Against Waterfall Design and The Supreme Acceptance of Agile Development, something was lost. Don't get me wrong, waterfall design takes things way too far. Huge systems specified in minute detail end up being the WRONG systems specified in minute detail. We were right to strike it down. But what took its place is too far in the other direction. Now we have projects with short, badly written, or entirely missing documentation. Some projects don't even have a Readme!

This is not acceptable. There must be some middle ground between reams of technical specifications and no specifications at all. And in fact there is. That middle ground is the humble Readme.

It's important to distinguish Readme Driven Development from Documentation Driven Development. RDD could be considered a subset or limited version of DDD. By restricting your design documentation to a single file that is intended to be read as an introduction to your software, RDD keeps you safe from DDD-turned-waterfall syndrome by punishing you for lengthy or overprecise specification. At the same time, it rewards you for keeping libraries small and modularized. These simple reinforcements go a long way towards driving your project in the right direction without a lot of process to ensure you do the right thing.

By writing your Readme first you give yourself some pretty significant advantages:

- Most importantly, you're giving yourself a chance to think through the project without the overhead of having to change code every time you change your mind about how something should be organized or what should be included in the Public API. Remember that feeling when you first started writing automated

> **“The Readme should be the single most important document in your codebase; writing it first is the proper thing to do.”**

code tests and realized that you caught all kinds of errors that would have otherwise snuck into your codebase? That's the exact same feeling you'll have if you write the Readme for your project before you write the actual code.

- As a byproduct of writing a Readme in order to know what you need to implement, you'll have a very nice piece of documentation sitting in front of you. You'll also find that it's much easier to write this document at the beginning of the project when your excitement and motivation are at their highest. Retroactively writing a Readme is an absolute drag, and you're sure to miss all kinds of important details when you do so.
- If you're working with a team of developers you get even more mileage out of your Readme. If everyone else on the team has access to this information before you've completed the project, then they can confidently start work on other projects that will interface with your code. Without any sort of defined interface, you have to code in serial or face reimplementing large portions of code.
- It's a lot simpler to have a discussion based on something written down. It's easy to talk endlessly and in circles about a problem if nothing is ever put to text. The simple act of writing down a proposed solution means everyone has a concrete idea that can be argued about and iterated upon.

Consider the process of writing the Readme for your project as the true act of creation. This is where all your brilliant ideas should be expressed. This document should stand on its own as a testament to your creativity and expressiveness. The Readme should be the single most important document in your codebase; writing it first is the proper thing to do. ■

Tom Preston-Werner lives in San Francisco and is a cofounder of GitHub and the inventor of Gravatars. He loves giving talks about entrepreneurship, writing Ruby and Erlang, and mountain biking through the Bay Area's ancient redwood forests.

Reprinted with permission of the original author. First appeared in *http://hn.my/rdd/*.

What Is LaTeX And Why You Should Care

By SLAVA AKHMECHET

Every time there is a link to a resource even remotely related to academia, it's available only in a weird format that looks like it was invented by Martians three thousand years ago while they were stuck on a strange planet light-years away from home. It's never something you can easily open – it's not HTML, not a Word document, not a text file. You're lucky if it's PDF. Most of the time it's PostScript or a mysterious DVI format that nobody outside of a select group of High Priests of Martianic Church knows how to open.

Reading the article ends up being a scavenger hunt for utilities found on obscure FTP mirrors of some .edu domains that end up having a user interface from Stone Age. While you search for these utilities you will undoubtedly see a few references to LaTeX which at first glance appears to be some document format invented by the aforementioned group of homesick Martians and requires ten pages just to explain what exactly it is it does. When you finally manage to open the file you're greeted by a word "abstract" instead of a word "summary" and as you try to scroll through the article you find that the scroll wheel abruptly stops at the end of the first page. You glance at the toolbar to find a "next page" button and see a row of various arrows with no immediately decipherable meaning. This is the final straw. You curse the ivory towers inhabitants in all known tongues for wasting ten minutes of your life, close the document without ever actually really looking at it, and go on with your life trying to block this painful encounter forever.

That's a pity. A little bit more persistence and you would have discovered a document processing nirvana.

PostScript

The first thing critical reading courses teach is that analyzing a piece of text involves analyzing its author's intent. Who is the author? What is the target audience? Why is the author writing to the target audience in the first place? If you're trying to understand a piece of writing, answering these questions is half the battle. For example, the target audience for a blogger is anyone who will listen. Bloggers tend to write to drive traffic as high as they can either to make money off advertising and affiliate programs, or to become famous, or, as yours truly, as part of a treacherous secret plot to take over the world.

Of course academic writers aren't bloggers. They couldn't care less about traffic or adsense. Most of the time they have too much on their minds to think about becoming famous. They're not even trying to take over the world. The only thing they dream about in the showers and in their sleep and on their way to work is getting published.

Whenever someone tries to go anywhere in academia beyond the undergraduate degree, they hear this phrase far more than they can handle without going insane. "I need to get published." "Are you published?" "Where is he published?" Published, published, published. The Holy Grail for anyone in academia is getting published in a prestigious journal in their field. If you're a graduate student, that's what you need to get a PhD someone except your mother will care about. If you already have a PhD, that's what you need to get a job as a professor in a good university. If you're already a professor, that's what you need to get and keep government grants for research. Even if you're an undergraduate, publishing an article nobody will ever read in a journal nobody has ever heard of can help you get into a good graduate program.

So, if you're an academic writer, you aren't going to write your article for the general public. You aren't even going to write it for fellow scientists. You're going to write it for people who hold your academic future in their hands – the journal editors. And journal editors are a very particular bunch that likes to receive submissions that adhere to strict guidelines.

I am not familiar with the dark underworld of journal publishing so I won't get into details, but the idea is simple. The work of the editors eventually ends up on the table of folks called the publishers. These are the people that take a piece of text, feed it into printing machines, get thousands of copies, and distribute them to subscribed recipients. The publishers couldn't care less about what they're printing – they just want to get it in a format that their printing machines understand. And this format isn't a Word document. Because publishers deal with huge volumes and very different types of documents, they want to receive them in a very specific format. A format that tells them exactly how and where to print every dot. They don't want to hear anything about paragraphs of text or tables of numeric values. They want to know how many inches from the left margin should the printer put the first dot and at what offset to put the next.

Now, back to the journal editors. Every day their mailbox contains dozens of submissions from every poor shmuck that wants the honor of being published in their periodical. They have to read through the submissions and only pick the best work to make sure they don't print something that doesn't make sense and make their journal look stupid[1]. The last thing they want to deal with is converting the submissions from whatever exotic format the authors decided to write them in to whatever peculiar format the published requires. So the journals set strict guidelines – you can only send submissions in PostScript or DVI (which incidentally turn out to be the formats their publishers accept).

Of course if you're Albert Einstein you can engrave your submission on a piece of rock, ship it to the journals via FedEx, and make them pay the bill. They'll be head over heels to accept it and do the format conversion work. But if you're Joe Mediocre, Ph.D., submitting your tenth paper in ten years on individual differences versus social dynamics in the formation of aquarium fish dominance hierarchies[2] to account for how you spent public funds granted to you by the NSF, you better submit your paper in PostScript. You know what'll happen if you don't. You won't get published this year, the NSF will take your grants away, you'll get kicked out of the faculty without tenure (why keep you around if you don't bring in any research money?), and you won't be able to unconditionally get university pay for the rest of your life without ever actually producing useful work.

LaTeX

These days there are add-ons for Microsoft Word that allow you to save your documents in PostScript. In the old days, when Word wasn't available, people used a format called LaTeX. It was a structured human readable format not unlike XML. People wrote their documents in text editors using LaTeX tags to specify sections, subsections, paragraphs, etc. After they were done with their document they ran it through a program that used stylesheets (conceptually not very different from CSS) to render a LaTeX document into another format (more often than not the end result was PostScript but it could just as easily have been HTML, PDF, DVI, etc.) Back then if you didn't like LaTeX you were forced to use it for a number of reasons. There were no other alternatives to generate PostScript files. Even if you could create one directly, different journals expected different formatting to fit their overall style. The only way to accommodate this requirement was to use LaTeX along with the stylesheets the journals provided[3].

Now that the old days are long gone and word processors come preinstalled with every machine, why should we care about this remnant of history? The answer is that remarkably LaTeX is much better suited for composing and distributing most types of documents than any other modern word processor on the market that I am aware of. Just like programming languages tend to converge towards Lisp because it got things right the first time around, so do the Word Processors tend to converge towards LaTeX.

Separation Of Markup And Presentation

When I started writing articles for defmacro, I did it in Microsoft Word. This was the word processor I've used since high school, throughout college, and at work. I saw no reason not to use it for writing articles for this website. I soon discovered that I'm not being very productive. It turned out that when writing documents that have valuable content – documents that cannot be written in a single evening and that people might want to read (unlike my college papers), Microsoft Word hindered me more often than it managed to provide assistance. Amazingly, I was far more productive writing articles directly in XHTML using Emacs (the best editor I've ever used[4]).

Aside from the obvious requirement to be able to efficiently edit text I needed my word processor to help me do two things: specify the structure of my document as I write it and let me style it later. Surprisingly Microsoft Word isn't very good for creating documents in this manner. While it supports styling and structural markup, it doesn't in any way encourage it. By default it's much easier to mark a selection as bold than to emphasize it using markup. XHTML, on the other hand, is different. I can only specify structure. If I try to use old HTML styling tags, it doesn't validate. This way I can focus on the content of my document and its structure. I can style it with CSS later. I can even provide different styles for my site, for printing,

Reprinted with permission of the original author. First appeared in *http://hn.my/latex/*.

and for other sites that might want to publish my articles.

It is common wisdom among programmers that information and the way it's presented should be separated. A well defined boundary between markup and styling allows to easily add other ways to present information. Additionally, it greatly enhances the ability to change information independently from its presentation. These are both very desirable properties and they are not limited to web pages. None of the mainstream word processors that I am aware of promote this paradigm. If I want to write documents this way I'm left with relatively few alternatives. XHTML and CSS are one, but they're relatively new technologies designed specifically for document distribution over HTTP. There is no easy way to convert my XHTML document along with appropriate CSS stylesheets to a single file I could send someone over e-mail. LaTeX does better. Once I create a LaTeX document I can easily convert it to any format I am interested in, including XHTML and Microsoft Word Document. I can compose documents the way I like and distribute them to the world in any format that happens to be fashionable at the time. As a bonus LaTeX has tags for almost everything I may want to specify in my documents. And if it doesn't, I can extend it with my own.

Modern office suites are already moving towards markup and styling. It will take them many years to embrace this paradigm completely and shed the legacy of styling interleaved with the document – a very poor design for obvious reasons. On the other hand, LaTeX is here today and there is no reason for us to to wait for word processors to catch up.

Open Document Format

For the past couple of years there has been a big debate sparked by the OpenDocument Format Alliance. Companies and governments decided they no longer want to be restricted to using Microsoft Word to edit and distribute their documents and came up with a radical idea that their information should be stored in an open format in order to allow competing word processors to have a real chance to win market share. Of course OpenDocument isn't here yet. Nobody can agree on the tags and Microsoft doesn't want to let go of market domination it has achieved by locking people into their format.

There is no reason for OpenDocument Format Alliance to reinvent the wheel and there is no reason for us to wait until they're done. LaTeX is already here. When you create your next document, let it rise to the occasion. The format is open and has a wide variety

of standard tags. It is human readable and can be modified in a multitude of editors from Notepad, to Emacs, to visual editors like Lyx. Additionally, LaTeX has a wide pool of available importers and exporters – you can import pretty much any document into LaTeX, modify it, and export it back into any format you like (from Word to HTML to PDF). LaTeX has everything an open, portable, extensible format should have. The only thing missing is the hype.

What's next?

Word Processors are the least useful components of modern office suits. An argument about Microsoft Word vs. Word Perfect is a false dilemma as there are better alternatives. Don't let LaTeX intimidate you. Once you play around with it and take some time to understand it, it becomes obvious that it's a very natural design – another proof that most great software was designed early in computer history. It may seem alien and dated but behind the cover there is a very powerful way to compose and distribute documents. Do a google search on LaTeX and you'll find plenty of tools equipped to edit LaTeX documents (this is somewhat like a multitude of HTML editors out there). Alternatively, if you don't feel like learning LaTeX tags, download Lyx – a visual document processor that takes care of the details behind the scenes. ■

Notes

1. I am, of course, referring to WMSCI 2005, a conference that accepted a paper generated by SCIgen, a random paper generator. Surely journal editors all over the world doubled their vigilance after this incident.
2. I'm not making this up. Really.
3. LaTeX is a de facto standard for publishing in academic journals. Most journals provide LaTeX stylesheets that allow you to format your paper according to specific requirements automatically.
4. One of the goals of this website is explaining the benefits of good technologies that are generally considered tricky to explain to the uninitiated (the examples I've already written about are Lisp and Functional Programming). In this sense Emacs fits right in. I hope to write an article about it some time in the future.

Slava has built technology for infrastructure software, consumer web, and financial companies. He is interested in high level programming languages, compilers, data storage systems, and software start-ups. If he had to eat one type of food for the rest of his life, it would be sushi. He is now on leave from the Ph.D. program in Computer Science at Stony Brook University, working on RethinkDB.

Don't Let jQuery's $(document).ready() Slow You Down

By DAVE WARD

JQUERY'S `$(document).ready()` EVENT is something that you probably learned about in your earliest exposure to jQuery and then rarely thought about again. The way it abstracts away DOM timing issues is like a warm security blanket for code running in a variety of cold, harsh browser windows.

Between that comforting insurance and the fact that deferring everything until `$(document).ready()` will never break your code, it's understandable not to give much thought to its necessity. Wrapping `$(document).ready()` around initialization code becomes more habit than conscious decision.

However, **what if `$(document).ready()` is slowing you down?** In this post, I'm going show you specific instances where postponing startup code until the document's ready event slows perceived page load time, could leave your UI needlessly unresponsive, and even causes initialization code to run slower than necessary.

Example: live()

One of the most popular uses for jQuery's `live()` is to maintain event handlers on elements that are dynamically created and destroyed over time. Instead of juggling traditional `bind()` handlers in response to those changes, `live()`'s event delegation allows you to declare handlers once up-front. Whether targeted elements exist at declaration time, or in the future, one `live()` handler will apply to them all.

Imagine that we have an application with several slideToggling sidebar blocks which may be dynamically added and removed while the user interacts with the page. You've probably seen `live()` used like this to simplify handling those future changes:

```
<html>
<head>
  <script type="text/javascript" src="jquery.js"></script>
  <script type="text/javascript">
    <!-- The sidebar event delegation is not registered
"here"... -->
    $(document).ready(function () {
      $('#Sidebar h3').live('click', function () {
        $(this).next().slideToggle();
      });
    });
  </script>
</head>
<body>
  <div id="Sidebar">
    <h3>Title 1</h3>
    <p>Text 1</p>

    <h3>Title 2</h3>
    <p>Text 2</p>
  </div>
</body>
<!-- ...but roughly down "here" -->
</html>
```

That usage is natural when you're hedging against AJAX-driven changes in the future. The volatility that you're concerned with won't happen until after the page loads, so it's intuitive to postpone worrying about them until after the document's ready event.

However, what would happen if you treated your HTML document's initial load process the same way as any other dynamic modifications?

```
<html>
<head>
  <script type="text/javascript" src="jquery.js"></script>
  <script type="text/javascript">
 <!-- The event handler is wired up "here"; immediately -->
    $('#Sidebar h3').live('click', function () {
      $(this).next().slideToggle();
    });
  </script>
</head>
<body>
  <div id="Sidebar">
    <h3>Title 1</h3>
    <!-- At this point, Title 1 is ready for action. -->
    <p>Text 1</p>

    <h3>Title 2</h3>
    <!-- Ditto for Title 2 at this point -->
    <p>Text 2</p>
  </div>
</body>
</html>
```

Not only does that work just as well as postponing the `live()` declaration until the document's ready event, but now the handlers are active during the page loading process. As the browser loads each <h3> element and adds it to the DOM, our click events are immediately ready to be handled.

Benefit: A more responsive UI

To see where the latter approach shines, imagine the page was very large and took several seconds to load, or that a script reference somewhere on the page was timing out. In scenarios like those, **jQuery's document ready event may not fire until considerably later than the targeted elements are visible to your users.**

```
<html>
<head>
  <script type="text/javascript" src="jquery.js"></script>
  <script type="text/javascript">
 <!-- The event handler is wired up "here"; immediately -->
    $('#Sidebar h3').live('click', function () {
      $(this).next().slideToggle();
    });

    <!-- This handler isn't active until... -->
    $(document).ready(function() {
      $('#Sidebar p').live('click', function() {
        // Important magic goes here.
      });
    });
  </script>
</head>
<body>
  <div id="Sidebar">
    <h3>Title 1</h3>
    <p>Text 1</p>
  </div>

  <script src="http://twitter.com/fail-whale.js"></script>
  <!-- ...way down here, *after* the script references
times out. -->
</body>
</html>
```

Why hold `live()` back until the document is ready? It doesn't matter if the selector matches any elements initially; they will immediately become active as they are rendered and appear on the page.

Benefit: Improved performance

A common criticism of using `live()` for event delegation is that it requires you to perform an initial selection of all of the elements that it targets. Since event delegation doesn't require any initial setup on each individual element, this pre-selection is a wasteful performance drag when there are dozens or hundreds of elements targeted.

However, if you register your `live()` handlers before those elements exist on the page, there is no performance penalty whatsoever. The event delegation can be registered very quickly, yet still works exactly the same as if you had waited until the ready event.

```
<html>
<head>
  <script type="text/javascript" src="jquery.js"></script>
  <script type="text/javascript">
    <!-- Fast. Runs before any of the TRs exist. -->
    $('#MyTable tr').live('click', function () {
      $(this).toggleClass('highlight');
    });

    <!-- Slow. Doesn't run until the table is rendered. -->
    $(document).ready(function() {
      $('#MyTable tr').live('click', function () {
        $(this).toggleClass('highlight');
      });
    });
  </script>
</head>
<body>
  <table>
    <!-- Hundreds or thousands of rows here -->
  </table>
</body>
</html>
```

Example: $.ajax()

Another situation where `$(document).ready()` may be holding you back is when you make an AJAX request immediately as a page is loading. Displaying recent Twitter updates is a common example of that:

```
<html>
<head>
  <script type="text/javascript" src="jquery.js"></script>
  <script type="text/javascript">
    $(document).ready(function() {
  <!-- $.getJSON() request to retrieve Twitter updates. -->
    });
  </script>
</head>
<body>
  <!-- A typically large page here -->
</body>
<!-- The Twitter request doesn't *begin* until here. -->
</html>
```

Even though the `$.getJSON()` snippet is located at the beginning of the page, it isn't executed until the entire page has loaded and the ready event has fired. **Why wait until the page is loaded in order to begin the AJAX request?**

```
<html>
<head>
  <script type="text/javascript" src="jquery.js"></script>
  <script type="text/javascript">
  <!-- $.getJSON() request to retrieve Twitter updates. -->
    <!-- The request begins immediately. -->
  </script>
</head>
<body>
  <!-- A typically large page here -->
</body>
</html>
```

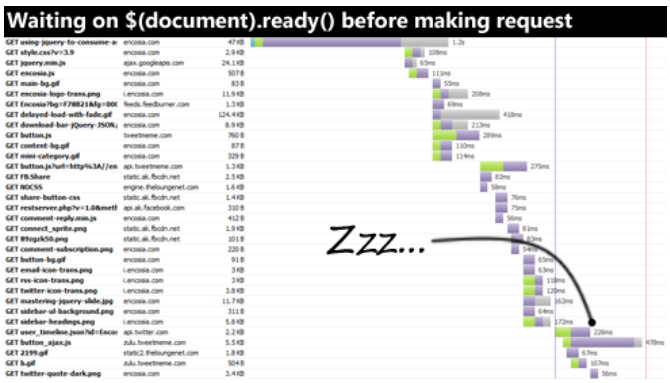
This is a nice improvement even when you're making a request to a local endpoint, but is even more beneficial here because third-party requests circumvent the browser's per-domain request limit. That third-party request to Twitter runs in parallel with the rest of the page's normal loading timeline.

Better yet, since the dynamic script element injection used in a JSONP request is asynchronous, there's no drawback to initiating the request early. Even if Twitter is slow or down (imagine that), the request won't drag the page down.

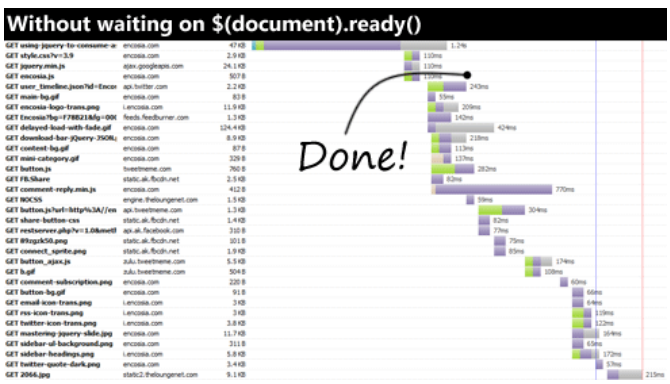
Benefit: Performance

To show you a visualization of how the previous two approaches differ, I used my own site as a guinea pig. First, I wrapped a `$.getJSON()` request to Twitter in `$(document).ready()` and placed it in the <head> of my site template.

This is how the site loads in that configuration, taken from Firebug's Net tab:



Now, here's the same `$.getJSON()` request, located in the same position in the <head> of the page, but without the `$(document).ready()` wrapper:



We just pulled a bit of performance right out of thin air. This isn't simply perceived performance, which is nice enough, but a truly faster overall load time.

Note: You might notice that the ready event came later in the second example and be concerned that it was due to the early `$.getJSON()` *request. It wasn't. If you look closely, a blocking <script> reference to one of the page's local scripts took unusually long in the second run, which pushed everything back about 700ms longer than usual.*

Reprinted with permission of the original author. First appeared in *http://hn.my/jquery/*.

Conclusion

I think the preceding examples are compelling, but I'm also not suggesting that this is appropriate in every case. Often, it's best to move every bit of JavaScript to the bottom of your pages (e.g. a public-facing, non-application site like this one). When your scripts are located at the bottom of the page, it doesn't matter whether you use `$(document).ready()` or not; everything is effectively running when the document ready event fires anyway.

However, when you're building the type of script-heavy "application" that behooves your placing script references in the document's <head>, keeping these ideas in mind can have a tangible impact on the performance of your application. ■

Dave Ward is an independent consultant in Atlanta, Georgia, specializing in creating functional, interactive web applications with HTML, CSS, and JavaScript. With over fifteen years' experience developing websites, he has been recognized as a Microsoft ASP.NET MVP and a member of the ASP Insiders. You can find him online at http://encosia.com or @encosia on Twitter.

SSH: Tips And Tricks You Need

By SYMKAT

SSH is one of the most widely used protocols for connecting to remote shells. While there are numerous SSH clients the most-used still remains OpenSSH's ssh. There is a plethora of tips and tricks that can be used to make your experience even better than it already is. Read on to discover some of the best tweaks to your favorite SSH client.

Adding A Keep-Alive

A keep-alive is a small piece of data transmitted between a client and a server to ensure that the connection is still open or to keep the connection open. Many protocols implement this as a way of cleaning up dead connections to the server. If a client does not respond, the connection is closed.

SSH does not enable this by default. There are pros and cons to this. A major pro is that under a lot of conditions if you disconnect from the Internet, your connection will be usable when you reconnect. For those who drop out of WiFi a lot, this is a major plus when you discover you don't need to login again.

For those who get the following message from their SSH client when they stop typing for a few minutes it's not as convenient:

```
symkat@symkat:~$ Read from remote host symkat.com:
Connection reset by peer
Connection to symkat.com closed.
```

This happens because your router or firewall is trying to clean up dead connections. It's seeing that no data has been transmitted in N seconds and falsely assumes that the connection is no longer in use.

To rectify this you can add a Keep-Alive. This will ensure that your connection stays open to the server and the firewall doesn't close it.

To make all connections from your shell send a keepalive add the following to your ~/.ssh/config file:

```
KeepAlive yes
ServerAliveInterval 60
```

The con is that if your connection drops and a KeepAlive packet is sent SSH will disconnect you. If that becomes a problem, you can always actually fix the Internet connection.

Multiplexing Your Connection

Do you make a lot of connections to the same servers? You may not have noticed how slow an initial connection to a shell is. If you multiplex your connection you will definitely notice it though. Let's test the difference between a multiplexed connection using SSH keys and a non-multiplexed connection using SSH keys:

```
# Without multiplexing enabled:
$ time ssh symkat@symkat.com uptime
 20:47:42 up 16 days,  1:13,  3 users,  load average: 0.00,
0.01, 0.00

real    0m1.215s
user    0m0.031s
sys     0m0.008s

# With multiplexing enabled:
$ time ssh symkat@symkat.com uptime
 20:48:43 up 16 days,  1:14,  4 users,  load average: 0.00,
0.00, 0.00

real    0m0.174s
user    0m0.003s
sys     0m0.004s
```

We can see that multiplexing the connection is much faster, in this instance on an order of 7 times faster than not multiplexing the connection. Multiplexing allows us to have a "control" connection, which is your initial connection to a server, this is then turned into a UNIX socket file on your computer. All subsequent connections will use that socket to connect to the remote host. This allows us to save time by not requiring all the initial encryption, key exchanges, and negotiations for subsequent connections to the server.

To enable multiplexing do the following:
In a shell:

```
$ mkdir -p ~/.ssh/connections
$ chmod 700 ~/.ssh/connections
```

Add this to your ~/.ssh/config file:

```
Host *
ControlMaster auto
ControlPath ~/.ssh/connections/%r_%h_%p
```

A negative to this is that some uses of ssh may fail to work with your multiplexed connection. Most notably commands which use tunneling like git, svn or rsync, or forwarding a port. For these you can add the option –oControlMaster=no. To prevent a specific host from using a multiplexed connection add the following to your ~/.ssh/config file:

```
Host YOUR_SERVER_OR_IP
MasterControl no
```

There are security precautions that one should take with this approach. Let's take a look at what actually happens when we connect a second connection:

```
$ ssh -v -i /dev/null symkat@symkat.com
OpenSSH_4.7p1, OpenSSL 0.9.7l 28 Sep 2006
debug1: Reading configuration data /Users/symkat/.ssh/config
debug1: Reading configuration data /etc/ssh_config
debug1: Applying options for *
debug1: auto-mux: Trying existing master
Last login:
symkat@symkat:~$ exit
```

As we see no actual authentication took place. This poses a significant security risk if running it from a host that is not trusted, as a user who can read and write to the socket can easily make the connection without having to supply a password. Take the same care to secure the sockets as you take in protecting a private key.

Using SSH As A Proxy

Even Starbucks now has free WiFi in its stores. It seems the world has caught on to giving free Internet at most retail locations. The downside is that more teenagers with "Got Root?" stickers are camping out at these locations running the latest version of wireshark.

SSH's encryption can stand up to most any hostile network, but what about web traffic?

Most web browsers, and certainly all the popular ones, support using a proxy to tunnel your traffic. SSH can provide a SOCKS proxy on localhost that tunnels to your remote server with the –D option. You get all the encryption of SSH for your web traffic, and can rest assured no one will be capturing your login credentials to all those non-ssl websites you're using.

```
$ ssh -D1080 -oControlMaster=no symkat@symkat.com
symkat@symkat:~$
```

Now there is a proxy running on 127.0.0.1:1080 that can be used in a web browser or email client. Any application that supports SOCKS 4 or 5 proxies can use 127.0.0.1:1080 to tunnel its traffic.

```
$ nc -vvv 127.0.0.1 1080
Connection to 127.0.0.1 1080 port [tcp/socks] succeeded!
```

Using One-Off Commands

Often times you may want only a single piece of information from a remote host. "Is the file system full?" "What's the uptime on the server?" "Who is logged in?"

Normally you would need to login, type the command, see the output and then type exit (or Control-D for those in the know.) There is a better way: combine the ssh with the command you want to execute and get your result:

```
$ ssh symkat@symkat.com uptime
 18:41:16 up 15 days, 23:07,  0 users,  load average: 0.00,
0.00, 0.00
```

This executed the ssh symkat.com, logged in as symkat, and ran the command "uptime" on symkat. If you're not using SSH keys then you'll be presented with a password prompt before the command is executed.

```
$ ssh symkat@symkat.com ps aux | echo $HOSTNAME
symkats-macbook-pro.local
```

This executed the command ps aux on symkat.com, sent the output to STDOUT, a pipe on my local laptop picked it up to execute "echo $HOSTNAME" locally. Although in most situations using auxiliary data processing like grep or awk will work flawlessly, there are many situations where you need your pipes

and file IO redirects to work on the remote system instead of the local system. In that case you would want to wrap the command in single quotes:

```
$ ssh symkat@symkat.com 'ps aux | echo $HOSTNAME'
symkat.com
```

As a basic rule if you're using > >> < – or | you're going to want to wrap in single quotes.

It is also worth noting that in using this method of executing a command some programs will not work. Notably anything that requires a terminal, such as screen, irssi, less, or a plethora of other interactive or curses based applications. To force a terminal to be allocated you can use the –t option:

```
$ ssh symkat@symkat.com screen -r
Must be connected to a terminal.
$ ssh -t symkat@symkat.com screen -r
$ This worked!
```

Making SSH A Pipe

Pipes are useful. The concept is simple: take the output from one program's STDOUT and feed it to another program's STDIN. OpenSSH can be used as a pipe into a remote system. Let's say that we would like to transfer a directory structure from one machine to another. The directory structure has a lot of files and sub directories.

We could make a tarball of the directory on our own server and scp it over. If the file system this directory is on lacks the space though we may be better off piping the tarballed content to the remote system.

```
$ ls content/
1   18  27  36  45  54  63  72  81  90
10  19  28  37  46  55  64  73  82  91
100  2  29  38  47  56  65  74  83  92
11  20  3   39  48  57  66  75  84  93
12  21  30  4   49  58  67  76  85  94
13  22  31  40  5   59  68  77  86  95
14  23  32  41  50  6   69  78  87  96
15  24  33  42  51  60  7   79  88  97
16  25  34  43  52  61  70  8  89  98
17  26  35  44  53  62  71  80  9  99

$ tar -cz content | ssh symkat@symkat.com 'tar -xz'
$ ssh symcat@symkat
symkat@lazygeek:~$ ls content/
1   14  2   25  30  36  41  47  52  58  63  69  74  8   85
90  96
10  15  20  26  31  37  42  48  53  59  64  7   75  80  86
91  97
100 16  21  27  32  38  43  49  54  6   65  70  76  81  87
92  98
11  17  22  28  33  39  44  5   55  60  66  71  77  82  88
93  99
12   18  23  29  34  4   45  50  56  61  67  72  78  83  89
94
13   19  24  3   35  40  46  51  57  62  68  73  79  84  9
95
```

What we did in this example was to create a new archive (-c) and to compress the archive with gzip (-z). Because we did not use –f to tell it to output to a file, the compressed archive was send to STDOUT. We then piped STDOUT with | to ssh. We used a one-off command in ssh to invoke tar with the extract (-x) and gzip compressed (-z) arguments. This read the compressed archive from the originating server and unpacked it into our server. We then logged in to see the listing of files.

Additionally, we can pipe in the other direction as well. Take for example a situation where you with to make a copy of a remote database, into a local database:

```
symkat@chard:~$ echo "create database backup" | mysql
-uroot -ppassword
symkat@chard:~$ ssh symkat@symkat.com 'mysqldump -udbuser
-ppassword symkat' | mysql -uroot -ppassword backup
symkat@chard:~$ echo use backup;select count(*) from
wp_links;" | mysql -uroot -ppassword
count(*)
12
symkat@chard:~$
```

What we did here is to create the database "backup" on our local machine. Once we had the database created we used a one-off command to get a dump of the database from symkat.com. The SQL Dump came through STDOUT and was piped to another command. We used mysql to access the database, and read STDIN (which is where the data now is after piping it) to create the database on our local machine. We then ran a MySQL command to ensure that there is data in the backup table. As we can see, SSH can provide a true pipe in either direction.

Using a Non Standard Port

Many people run SSH on an alternate port for one reason or another. For instance, if outgoing port 22 is blocked at your college or place of employment you may have ssh listen on port 443.

Instead of saying ssh –p443 you@yourserver.com you can add a configuration option to your ~/.ssh/config file that is specific to yourserver.com:

```
Host yourserver.com
Port 443
```

You can extrapolate from this information further that you can make ssh configurations specific to a host. There is little reason to use all those –oOptions when you have a well-written ~/.ssh/config file. ■

SymKat is an avid cook, Perl hacker, & Linux sysadmin with experience from major web hosting and content delivery companies who currently resides in Los Angeles, California.

In Praise Of Quitting Your Job

By BEN PIERATT

Reprinted with permission of the original author. First appeared in *http://hn.my/quitjob/*.

I WROTE THIS EMAIL to a friend a few weeks ago, and then the topic came up again last night with an old buddy who was frustrated with his work. He seemed to appreciate what I had to say, so I figured it might be worth sharing:

Thinking about your comment at the end our call. Thought I'd put some words down. Apologies in advance for the presumption.

The reason I'm so supportive of you quitting your job is that I'm intensely empathetic to your situation and I believe that you're doing everyone a disservice by sticking around.

I've worked for a handful of companies over the course of the last 6 years. I started all of them with a fair amount of enthusiasm, but within 5 months of each I dipped into a depression. By 7 months the work was having a tangible effect on my mood and outlook, and by nine months, I've quit almost every job I've held. The longest was 12 months at [Redacted], and that was only because I wanted my options to vest. I handed them my resignation on my 366th day.

I always feel like a waste of space in these situations. Part of the depression stems from being so useless. Why do I hate this job so much? What is wrong with me that I'm so entitled? People the world over have jobs they don't like, why am I unable to stick this out?

I could wax on this for a while (and I did, but then deleted all the paragraphs), but I think it comes down to the fact that, for some people, work is personal. Personal in the same way that singing or playing the piano or painting is personal.

As a creative person, you've been given the ability to build things from nothing by way of hard work over long periods of time. Creation is a deeply personal and rewarding activity, which means that your Work should also be deeply personal and rewarding. If it's not, then something is amiss.

Creation is entirely dependent on ownership.

Ownership not as a percentage of equity, but as a measure of your ability to change things for the better. To build and grow and fail and learn. This is no small thing. Creativity is the manifestation of lateral thinking, and without tangible results, it becomes stunted. We have to see the fruits of our labors, good or bad, or there's no motivation to proceed, nothing to learn from to inform the next decision. States of approval and decisions-by-committee and constant compromises are third-party interruptions of an internal dialog that needs to come to its own conclusions.

Your muse can only be treated as the secretary of a subcommittee for so long before she decides to pack up and look for employment elsewhere. If you aren't able to own the product and be creative, then you aren't able to do your work, and if you're not doing your work then you're negating a very real part of your personality, which is no good for anyone. No good for you and certainly no good for your employer.

I've come to terms with my own inherent work issues simply by recognizing that my weaknesses in one context are strengths in another. When I am able to own a project or product, I work hard and I work well, and I like to believe it shows in the results. Not everyone can do this. Not everyone is willing to spend stupid amounts of hours on a project simply because they believe in it. This is worth recognizing.

My point is simply this. From what little I understand of you and your situation, I feel like I can empathize. I would guess that you're juggling a handful of self-loathing with a justified sense of entitlement. This is something that I came to peace with after I left my last job, and I get the sense that you're still struggling with it.

I suspect that eventually our culture will catch up with our evolving understanding of work ethic and the personal nature of work in creative fields. In the meantime there's going to be a lot of wasted talent pushing too much effort in the wrong directions. It is clear to me and anyone who interacts with you that a misplacement of your energies is at everyone's loss. I hope that you're able to recognize this fact and move forward accordingly. ■

Ben Pieratt is a graphic designer living in Boston. His projects include Svpply, Lookwork, and the Egotist Network.

Design For Hackers: Why You Don't Use Garamond On The Web

By DAVID KADAVY

Amongst designers – especially print designers – Garamond is considered one of the best fonts in existence. It's timeless, and very readable. But, because of the limitations of current display technologies, it's not a good font to use in web copy – even with the advent of font embedding methodologies such as TypeKit and Google Font API.

One of the most important principles behind every good piece of design is that the designer has to master his or her medium. With any medium – whether it's pencil and paper, steel and glass, or pixels – the designer has to work with strengths and limitations. Work with these characteristics, and the design stands a chance to be good – work against them, and there is no chance.

Apple's lead designer, Jonathan Ive knows this. He recently said:

> *The best design explicitly acknowledges that you cannot disconnect the form from the material – the material informs the form…*

Medium and Form in Type History

Typography is the perfect vehicle with which to illustrate this concept throughout history. From the beginning, the forms of our letters have been influenced by the tools we used to create them.

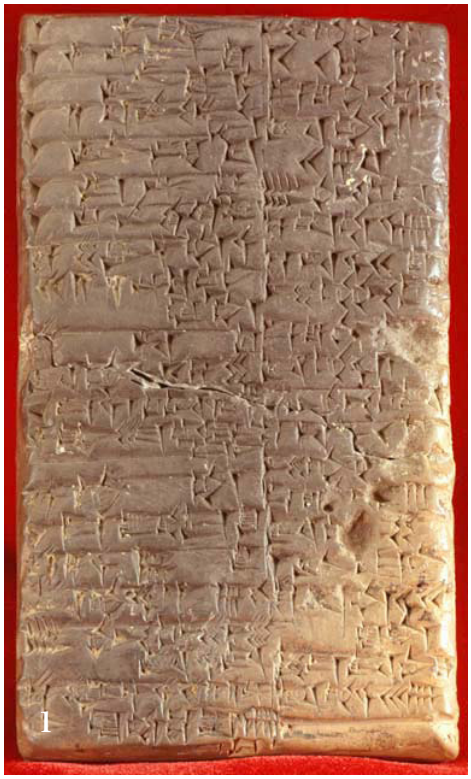
This cuneiform[1] inscribed tablet is an early example of how medium influenced form in written communication. You can see, looking at these pictograms, that they are made up of a series of indentions that are pretty much identical. This is because they were formed using a wedge-shaped stylus.

As this language was replaced in the west by our current roman characters, and the tools which we used changed, so did the form of our letters. Some of the best examples of early typography using roman characters are from – you guessed it – the Roman empire.

This is graffiti[2] from the ancient city of Pompeii. It was created using a brush, and this is apparent in the letterforms. You can see there's a great deal of variation in the strokes that make up the letters, and they all terminate with a soft point, just like you would expect from a brush.

Here's a picture[3] I took from Pompeii that I blogged about several years ago – dating back to the same time (remember, this city was frozen in time when it was buried under volcanic ash in 79AD). Only this time, the sign was chiseled in stone – and you can see how this has influenced the letters: all of the strokes of the letters are uniform in width, and to make the ends of those strokes looks nice – serifs were added. You can see little spur serifs from where the chisel was applied perpendicular to the stroke of each of these letters.

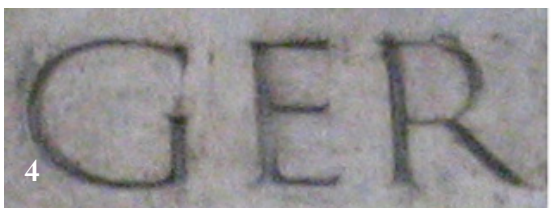
Now, moving more quickly through history, we have letters from the column of Trajan[4] (which inspired today's Trajan font), which were formed first by brush, then by chisel (it would have been awkward to chisel letters like the brush-drawn ones in the earlier Pompeii example). Then we moved on to lead and wood-cut printing, which first imitated work done by scribes with pens.



The characters on this cuneiform tablet are similar to one another because they were created with the same tool

This graffiti was clearly created with a brush

The forms of these letters were influenced by the chisel that they were created with



The lettering on the column of Trajan were brushed on, then chiseled

Once actual drawing tools were a smaller part of the design equation, typographers started to get more theoretical with their designs – creating constraints of their own – fonts like Bodoni are geometrically rationalized, as they were created in a medium (cast metal) with relatively few restrictions.

A Little Too Much Freedom?

In modern web typography, we still have the restriction that the letters of our alphabet take certain forms, but many restrictions have been removed. Rather than only having a couple of fonts available in our typecases, there are thousands. So, this makes it easy for bad habits to develop, such as trapping our information in images, or using fonts that just aren't good for the web.

So, what makes a font bad for the web? There's the widely-known issue of availability of fonts on the computers of our audience members – this, of course, is why we're usually using widely-available fonts like Arial, Verdana, Georgia, Times New Roman etc.. Now there are some pretty feasible ways of using whatever fonts we want – methods like SIFR, Typekit, and Google's new Font API, but that still doesn't mean you should use just any font. Even great classics like Garamond can be a disaster on the web, so its better to use a modern font that has been drawn with the screen in mind.

And the reason behind this is that our display technology isn't up to par with paper. You can see here a comparison of the great classic font, Garamond, blown up (as it might look on paper), next to a detail of what it would be anti-aliased at 12px height on a modern computer screen. You can see that it doesn't look so good on-screen, because it's just made up of a bunch of blocks of color.

The Fox

The letterforms of Bodoni are geometrically rationalized

What a 12px Garamond character looks like, blown up

Pompeii graffiti photo from virtusincertus, *http://www.flickr.com/photos/virtusincertus/*. Trajan's Column photo from Silver Tusk, *http://www.flickr.com/photos/silver_tusk/*

Working With the Screen

So, the popular web fonts (Arial, Verdana, Georgia, and Times New Roman) are such not only because of their wide availability, but because they are drawn with the screen's limitations in mind.

This Flash animation: *http://www.kadavy.net/experiments.html* that I created illustrates how pixels distort curvilinear form – such as that of typography. It's the same series of concentric rings, but as it changes sizes, you can see that a moiré effect results from trying to draw these rings out of mere pixels. So, the most web-appropriate fonts are drawn with these limitations in mind.

This illustration shows just what I mean by that. Georgia reads better on screen than Garamond primarily because it has a higher x-height (the height of an "x"), and – as a result – a larger eye. This prevents letters such as "e" from becoming muddled and unreadable, and overall makes the letters actually look larger. The notes on this illustration are in 9px Verdana with no anti-aliasing; and you can see those letters read very crisply, as this font was made for such an application.

Georgia has a huge advantage over Garamond on-screen because it was designed to be displayed as such from the very beginning, when it was designed by Matthew Carter for Microsoft in the mid-90's. This has manifest itself in sharp serifs on Georgia, rather than more subtly modeled ones on Garamond. Look at little curve on the bottom of Garamond. This gets blurred at smaller sizes, and hurts the legibility of Garamond.

This limitation of screen technology has been embraced, and taken to extremes, though.

Starting in the late 90's and early 00's, we saw lots of pixel fonts being used in Flash, such as these from Craig Kroeger's miniml.com, which are designed to be used at specific sizes, with no anti-aliasing.

Georgia is more readable than Garamond on-screen because of its larger x-height

When it was more common for computers to have only 256 colors, which caused dithering, designers embraced that constraint to inform their designs. Though ostensibly created to minimize bandwidth (another constraint of medium), designs that were created for the5k embraced dithering and lucidly used every pixel.

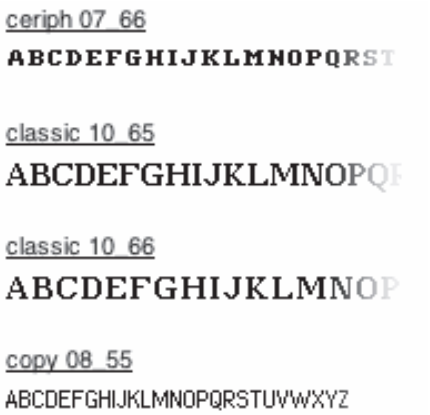
The "Web 2.0" design trends of the last five years or so, are thanks to display quality and bandwidth improving, removing some of this constraint. In 2000, 12% of web users had only 256 colors on their monitors – in 2010, 97% have over 16 million colors (the number of colors available has a big impact on how crisply type, images, or *gradients* are displayed). This has put into the hands of designers a color palette beyond that of CMYK printing, with increased bandwidth to push it through.

Additionally, displays are cramming in more pixels per inch (ppi). The cheap Dell monitor I'm typing this on is displaying at 100ppi, and my MacBook Pro is displaying at about 115ppi. Compare that to the iPhone 4, which displays at an impressive 326ppi. Now, we're starting to get some display technologies that are approaching the quality of paper when it comes to displaying letterforms readably.

So, maybe some day Garamond can make its comeback. ■

David Kadavy is a freelance Designer, and President of Kadavy, Inc. Though based in Chicago, his clients include the stars of Silicon Valley, such as oDesk, UserVoice, and PBworks.

The sharp edges of the serifs on Georgia make them display more crisply on-screen



Some type designs, like these from miniml.com, embrace the limitations of the pixel

The limitations of the pixel spawned design methods, such as the dithering used in this design

Reprinted with permission of the original author. First appeared in *http://hn.my/garamond/*.

Keep Calm And Carry On: What You Didn't Know About The Reddit Story

By ALEXIS OHANIAN

Few people outside of friends & family knew about the following at the time it was going on; bringing it up now, long after I've left reddit, feels less self-serving and will hopefully be instructive. This came up briefly in a talk I gave at MIT, but this feels a lot more comfortable to write than to speak about.

Steve and I spent every waking hour (and some dreaming, no doubt) after graduation with reddit somewhere on our minds. The time we spent working on it together only reinforced the marriage metaphor everyone uses about cofounders.

My life – and thus Steve's – was dramatically changed during those startup months for reasons beyond my control. I've lived a ridiculously fortunate life, so I knew it was only a matter of time before something was going to knock things a bit off course; I just didn't think it'd happen like this.

Just a month after we started working on reddit, Steve and I were wrapping up a game of WoW around 4am. I'd only been asleep for an hour when my cellphone rang.

My girlfriend's mother was on the phone. Her daughter had been studying abroad in Germany, was due home in just a couple weeks, and was now in the hospital. She'd fallen out of her apartment window. Five stories.

I spent a good part of our YC summer in Germany beside her hospital bed. Her mother remained until December when she finally came home after months of coma, surgeries, recovery, and rehab. (It's worth noting that German taxpayers kindly paid for every day of this world-class medical treatment. Danke.)

I can't stress what a tremendous recovery she's made. I had the honor of attending her graduation from the University of Virginia. Although we're no longer together, she remains someone who consistently inspires me.

Keep calm, carry on.

Little did I know, a couple months after my girlfriend's fall, I was due for another call.

My mom called me one Monday morning in September. She was distraught. Max, our family dog, had just died. Poor boy had been fighting Cushing's Syndrome for quite some time; my mom found him that morning in great distress and rushed him to our vet. There weren't very many options.

The most humane thing to do was euthanasia. I never got a chance to say goodbye to the good boy, but I take solace knowing he was with my mother, who doted on him like a son once I was out of the house.

It was hard on all of us, but it was hardest on my mom.

They were supposed to leave that evening for a trip to Norway. They'd planned it for months.

So I was surprised to get a call from my dad that evening (when am I going to learn to stop taking out-of-the-blue calls?).

He and mom were in the hospital. In hindsight, her anguish is possibly what triggered the seizure she had that afternoon, which led to the MRI that canceled their vacation.

My mother was diagnosed with a class IV Glioblastoma multiforme. Such an ugly name. I remember the first time I googled it, hoping I could search my way to a cure. But it basically meant terminal brain cancer. She was 51 when she was diagnosed.

I flew down to Maryland first thing the next morning. And you know the first thing she told me?

"I'm sorry. Sorry because I know how much you've already been through."

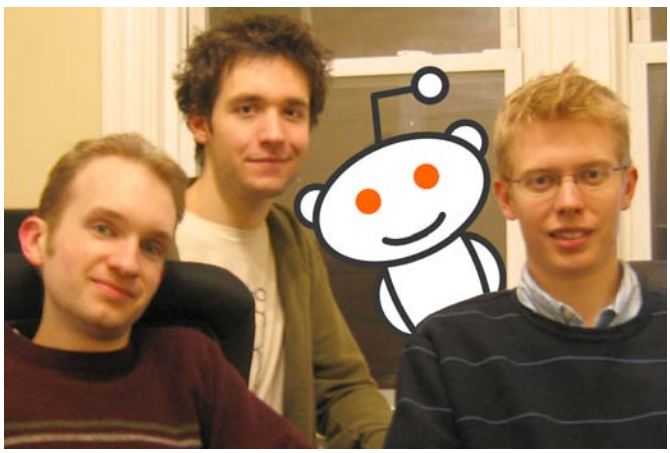
Keep calm, carry on.

During the next few years I spent a lot of time travelling between Boston (where reddit was based) and Maryland (where my parents lived). Every time I left her side, I was energized by her courage and unflagging spirit. She gave me all the inspiration I needed to wake up every morning and kick some ass, because that's what you have to believe as a startup founder.

If you've worked with the spineless, you know how frustrating it can be to deal with their poisonous helplessness – something that's only heightened in a startup where the most important thing you can do is not give up. And you'd better fucking believe that when you come home to a mother battling brain cancer and a father spending every waking hour taking care of her and running his own business, you don't complain, you don't cower, and you most certainly don't quit.

She fought for far longer than any doctor expected and died on March 15, 2008. But I got to prove that her 25 years of wholeheartedly supporting me weren't in vain – you can bet that had a lot to do with my feelings about selling reddit.

There were some dark months there, like living in the middle of an interminable fog. Upon reflection, I was probably suffering from depression for most of that startup. If you happened to meet me during that time, you probably wouldn't have known it.



But I got through it thanks to having a startup (and working with people like Steve & Chris):

- Freedom to travel whenever and wherever (I must've explained my 3G modem to every single nurse at Hopkins & NIH).
- It was something I could wholly invest myself in to keep my mind off everything else knowing that everything I was putting into it wasn't benefiting my boss.
- Having partners like Steve Huffman and Chris Slowe who never questioned what I was doing with my time, were absolutely supportive, and could always be counted on for a game of Soul Caliber or round of beer when I needed it. I hope I was at least half of all those things in return.

(I also got a lot of therapy from doodling all those alien logos for random holidays and events – it was something I knew she'd check every day. But that's certainly not for everyone. I started a photoblog for her to check regularly, too: OMGbabies. Cute baby animal photos are endorphin-tastic!)

Having been through all this, I can confidently say that starting a startup was the best thing that could've happened to me. Enduring all of that in an office job or law school would've been overwhelming.

Plenty of you reading this have no doubt been through the same or worse (and I wouldn't wish it on any of you who haven't) but know that under the right circumstances, having a startup could be extremely beneficial for your mental health.

As if you needed one more reason why you ought to start a startup.

Thank you, mom. I love you. ■

Alexis co-founded reddit, breadpig (LOLmagnetz, xkcd: volume 0, & more geekery), is an angel investor, and Kiva Fellow. He loves falafel & he'd like to make the world suck less.

On: How To Read Math

From PETER COOPER (petercooper)
If you like this, you might enjoy this delightfully funny, yet effective, introduction to calculus book from 1914: *http://hn.my/calculus/*.

It starts by ranting about uppity mathematicians and academics while showing how simply you can get your head around basic calculus.

On: Why Free Plans Don't Work

From ELI JAMES (shadowsun7)
I loved this article, but after thinking about it for a bit I realize that there *is* an argument for going free. Two pretty strong ones, in fact.

1. If you want to get acquired – and we're talking about a big acquisition here – it would do to go free so you manage to score a big-enough user base to get noticed. And so the bigger company is interested in acquiring you not only for your technology, but also for your users. (That said, acquisition may be a risky bet.)
2. Going free also makes sense if you want total market dominance. In which case, free is really the only way to go. The writer is speaking from the POV of a micropreneur, and it makes sense for him to retract his free plan (in fact, it makes sense for 37signals/ the 37signals model to do so too).

But if you're in the position of being the next Facebook, going free and dominating the market is really the best thing to do. (Or Youtube, or Google, or Flickr).

You find lots of users, lock out the competition, and *then* you can figure out how to monetize.

Which only goes to show that there are all kinds of businesses out there, and the advice you read on the net really should be done through the context of your particular business and/or market.

On: Keep Calm And Carry on: What You Didn't Know About The Reddit story

From ED WEISSMAN (edw519)
After focusing so much on the ones and zeroes, posts like this snap us back to all that really matters: other people.

In the past year, I have made dramatic changes in my life, both personal and business, for one reason: so that I can spend time with my mother who is suffering from severe dementia. We watch Jeopardy and Wheel of Fortune every night together. I yell out all the answers and she laughs, not caring whether they're right or wrong.

Before she started slipping away last year, she told me 2 things:

"From the moment I first saw you, I knew I would love you unconditionally forever."

and

"I'm so proud of you."

Everything else from this point forward is gravy.

Thank you, Alexis.

On: Design For Hackers: Why You Don't Use Garamond On The Web

From EDD SOWDEN (edd)
What this is actually saying is don't use Garamond for copy text as its not designed for current DPIs that are currently used on monitors. It doesn't mean you can't use it for titles or headings or devices with a high DPI.

On: What Is LaTeX And Why You Should Care

From ERICH HEINE (sophacles)
How appropriate, I just finished a paper today and submitted it for review. LaTeX is awesome and sucky at the same time. On the awesome side, you have structure separate from formatting. This comes with lots of benefits as mentioned in the article. It also allows you to use the concept of imports (or includes if you will), so you can have a *well factored paper*. It also allows you to do sane things like footnote, cite and reference diagrams and sources without a need for explicit number tracking – just give everything a unique identifier that works for you. Finally if you do well enough with your structuring you can output not just different file formats like pdf or ps or whatnot, you can also output completely different styles. It is pretty simple to wrap your core with the trappings of IEEE style for one version and a book-like style for another.

On the sucky side, the toolchain is notoriously difficult and cryptic. For some reason you have to make multiple passes of various tools by hand (or with a makefile – I recommend *http://code.google.com/p/latex-makefile/* it just works). The syntax can be a bit inconsistent. The worst is the errors tho, sometimes it is impossible to figure out why all your figures are showing up at the end of the document instead of in-place, or why all your references are failing to point at anything.

Overall tho, it is a fantastic system :)

On: Rich Programmer Food

From MAHMUD MOHAMED (mahmud)

Pretty entertaining, if a bit melodramatic.

I really wish people didn't mystify such a basic programming skill. Compiler hacking is something reserved for the wizards *only* if you take the classic definition of compiler implementation: an expensive engineering project, targeting a new processor, architecture or OS.

In that sense, sure. You will be working in a lab with hundreds of developers, and tinkering with a piece of multimillion dollar prototype.

In reality, however, "compiler construction" boils down to foundational language theory, along with various tricks and techniques for translating a set of very straightforward algebraic rules, to another set. Anytime you write regexes or XPath to extract a set of fields from a documents and transform to something "readable", you're almost writing a simple one pass assembler, using some implicit "business rules" (i.e. all numbers should be floats, names capitalized, etc.) for productions.

Compiler skills will give you the theoretical backbone to discover, refine and assess those implicit rules. Not to mention techniques for transforming them from one form to another.

To the list of skills made mystical and magical by people I would add Lisp. It's not magic. I mention it because it just so happens to have compiler construction lore aplenty.

The first Lisp exercises you will read in your lisp book of choice (often requiring nothing more than basic English literacy and 6th grade arithmetic) are the expression simplification and evaluation exercises. Transforming your elementary school algebra rules (multiplication and division before addition and subtraction, etc.) to something the machine can execute. The hardest part is just understanding the words: if you have a hunch for what "expression", "term", "rule", "left/right hand-side" and "precedence" might mean, you're good to start.

Few chapters of a Lisp book will spare you volumes of traditional compiler construction techniques, taught by rote methods.

The first time I attempted to read SICP I had convulsions and physical pain. The whole time I had this inferiority complex nagging at me, telling me this was something for "smart kids" and I was unworthy. But this stopped after I went through the first few parts of chapter 1, and took in the playful tone the text. I felt stupid afterward; like being afraid of a St. Bernard. It looks big, but it's actually bubbly.

Don't listen to people when they say something is difficult or not for the faint of heart. Put a saddle on Falkor and go flying!

On: In Praise Of Quitting Your Job

From JAMES KING (agentultra)

I just came out of a 2.5 year stint. Before that I went into consulting because I couldn't hold down a job for even a year. I figured it was boredom and thought the solution was to simply expose myself to a constant stream of new problems. Turns out it was simply *ownership* that was the problem. I never felt in control.

By *ownership* I mean ownership of my domain (area of involvement, etc) in the project. I don't mean ownership as in control and possession. The kind of ownership that kept me around at my last gig was the kind that let me make suggestions, criticisms, and decisions that were taken seriously. I had responsibility to back up every claim I made and that responsibility kept me highly motivated to produce the best software I could. It made the project feel more collaborative and kept me involved as a part of its development.

What kills that motivation is a loss of that ownership. In the final months of my last gig we brought on someone who took *total* ownership of the project practically from design to implementation. It no longer felt collaborative. I felt like a monkey in a room of monkeys trying to type out Shakespeare; as if I could replace myself with a junior programmer at half my salary and things would still run smoothly. That's not a good feeling and such loss of ownership (or lack of it in the first place) is completely demoralizing.

I get the sense that the OP was referring to this kind of ownership. The kind that makes you feel involved and responsible.

But does that mean you should quit your job? I don't think so. Some jobs will have ups and downs. I didn't leave my job when they brought the new guy on. I was going to stick it out... just circumstance brought my tenure there shorter than anticipated. I think you can stick it out in this way as well and avoid "depression." I take pride in my work and it does affect me very personally.. but you have to keep things in perspective. Especially when you have other people relying on you to keep your job.

Dream. Design. Print.

MagCloud, the revolutionary new self-publishing web service by HP, is changing the way ideas, stories, and images find their way into peoples' hands in a printed magazine format.

HP MagCloud capitalizes on the digital revolution, creating a web-based marketplace where traditional media companies, upstart magazine publishers, students, photographers, designers, and businesses can affordably turn their targeted content into print and digital magazine formats.

Simply upload a PDF of your content, set your selling price, and HP MagCloud takes care of the rest—processing payments, printing magazines on demand, and shipping orders to locations around the world. All magazine formatted publications are printed to order using HP Indigo technology, so they not only look fantastic but there's no waste or overruns, reducing the impact on the environment.

Become part of the future of magazine publishing today at www.magcloud.com.

25% Off the First Issue You Publish

Enter promo code **HACKER** when you set your magazine price during the publishing process.

Coupon code valid through February 28, 2011.
Please contact promo@magcloud.com with any questions.

MAGCLOUD