The Lisp Curse

By **Rudolf Winestock**

HACKERMONTHLY

Issue 13 June 2011

Curator
Lim Cheng Soon

Contributors
Paras Chopra
Rudolf Winestock
Thomas Buck
Udo Schroeter
Kenneth Myers
Howard Yeh
Ryan Tomayko
Angus Croll
Laurence Tratt

Proofreader
Emily Griffin

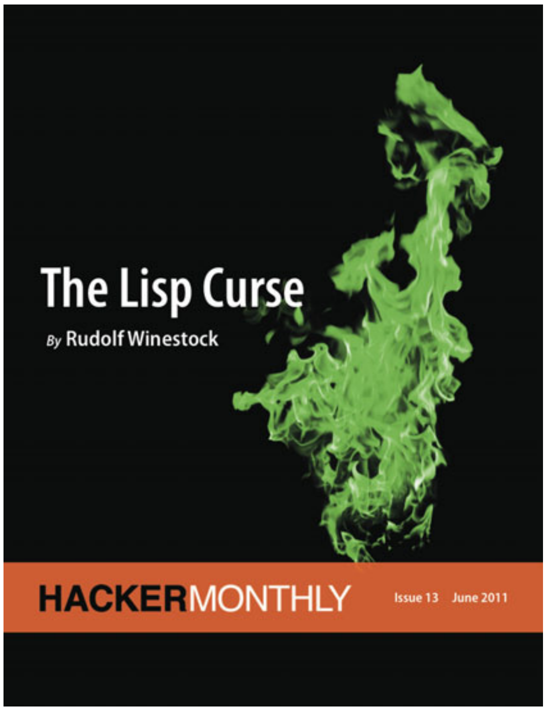
Printer
MagCloud

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

Advertising
ads@hackermonthly.com

Contact
contact@hackermonthly.com

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



Contents

How to Find Startup Ideas that Make Money

By PARAS CHOPRA

If your aim is to make money, pursuing such ideas can be risky. While idea-driven startups rarely make money, I professed a market-driven approach for someone looking to find startup ideas that actually make money.

Market-driven approach to finding startup ideas that make money

The market-driven approach is quite simple. It essentially means:

> *Find a startup idea that a) is already making money for someone else in a growing industry, b) interests you, and c) aligns with your skill sets. Once you find such an idea, simply carve out a niche within the industry by a) addressing pains of an under-served segment within that industry, or b) making it much easier to use than existing solutions, or c) disrupting the market by making your product accessible to masses at a much affordable price. And once you dominate a particular niche, expand from your niche with your eyes set on the largest player in the market.*

There is a lot going on here, so let's break it up.

Finding a startup idea

For most entrepreneurs, this is perhaps the most difficult phase of initiation. I have known people who would wait for years for that golden startup idea to strike. Truth is, even if you wait for years, startup ideas that are born out of a vacuum almost never work. As Steve Blank says, "no business survives first contact with the customer." So, why not skip the whole idea-game altogether and simply go ahead with ideas that other people have tried-and-tested? This is what market-driven approach is all about. Pick a growing market and simply make a better product.

Here are some essential ingredients of a market-driven startup idea:

- **Growing industry:** this is important because a rising tide lifts all boats. Also, a growing industry means that most probably a strong

“The key idea here is to find an industry where you know people are making money.”

leader is yet to be established, and the field is open for many new players, one of whom could be you. How to find industries that are growing? One good resource is Inc’s 5000 fastest growing companies list. In that list you can find companies that have been growing at +1000% every year for the last 3 years and have revenues in millions of dollars. If they can do it, why can’t you?

- **Industry that interests you:** aim is to not just make money but have fun on the way, right? Hence, it is important to pick a startup idea in an industry that appeals you. Even though eCommerce industry for ladies’ bags and purses might be growing, if you don’t see yourself passionate about it, don’t pick it!
- **Industry where you have a chance:** it is bit obvious, but there are a lot of things in life that appeal to us, but we’ve got no chance (for geeks: most obvious example is dating a hot lady!). For example, it goes without saying that even if the machine vision industry is growing and people are making money licensing such technology, if it requires a PhD and you don’t have it, it is probably not worthwhile to pursue an idea in that industry.

The key idea here is to find an industry (like SEO, document management, enterprise productivity, eCommerce for travel, etc.) where you know people are making money. Inc 5000, Mixergy interviews and Flippa.com are just some of the sources where companies reveal how much money they are making. Make a list of industries that make money for other people, appeal to you, and are relevant to your skill set. Finally select any one of them (though in most cases you will end up with only 1 or 2 which satisfy all 3 criteria). Don’t be ashamed of this activity, as we are not “copying” business ideas; we are simply using information to select which industry your startup should belong to.

$5700, *flickr.com/photos/amagill/362201147/*

“Lack of competitors in the market is a serious indicator that nobody has found it profitable.”

But there are competitors in an established market!

That’s precisely the key to this approach. Lack of competitors in the market is a serious indicator that nobody has found it profitable. So, you would want to pick a startup idea that has competitors. In addition to signaling that a market is profitable, competition also helps in positioning your startup. When you are new, nobody understands your offering and frankly nobody has time and patience to understand it. They are simply too busy to digest an entirely new idea or product offering. However, when you position it against established competition, you instantly have their attention and they instantly understand the differentiation. Now customers don’t have to understand new concepts, they simply understand what’s so different about you.

This strategy of positioning against established competition is very powerful. That’s why when cars were invented, they were first called horseless carriages. And that’s why I have positioned my startup Visual Website Optimizer [visualwebsiteoptimizer.com] as a much easier alternative to Google Website Optimizer with all the features of Omniture Test and Target. (You may not understand the positioning, but my target market of people who do A/B testing day-in and day-out would instantly get it.)

Even with all the benefits, many entrepreneurs still fear established competition. In the previous step, once you pick an industry that you want to start with, find a niche which you can dominate initially. It is important to become a leader in at least one aspect of your industry. That aspect can be:

- **Serving an under-served segment.** Imagine you picked SEO as an industry, next step is to do research (hint: talking to people works best, but probably I will address this in next blog post) on what the current pain points are that are not addressed by existing solutions (including the market-leading one). It may be the case that only a small segment is unhappy, but in a growing market even that small segment can be pretty large (in terms of revenue potential) for a startup. So, for example, you find that marketing agencies want a white-labeled solution for their clients. There, you have a startup idea: white-labeled SEO tools for agencies. Similarly, if it is document management, you may find that most solutions are so generic that a specific subset of market like accountants are craving much better management of Excel spreadsheets. So, there you have another startup idea: document management for accountants and financial planners. (Warning: the two startup ideas above may or may not work. They are figments of my imagination with no market research!)

“It is important to carve out a niche that you can dominate with your startup.”

- **Another differentiator of your idea could be usability and ease of use.** Most likely, customers in any industry are fed up with existing, bloated solutions with hard-to-use interfaces. Simply pick an industry and make it drop-dead easy to use. People usually drastically under-estimate how big an advantage ease-of-use can be for a startup. However, simply look at some examples. File sharing existed before Dropbox. Social networking existed before Facebook. A/B testing existed before (my) Visual Website Optimizer. What all of these products did was to dramatically simplify the key activity in an industry. You can do the same. Taking example of SEO, make a product that makes it a no-brainer to generate new content and build backlinks for it. Make it so simple that even a 5th grader can do it, and you have a winner.
- **Disrupt an industry with a lower (entry) price point.** If your industry is growing and existing solutions are exorbitantly priced, there may be an opportunity to build a product as great as the leading one in the market by simply providing it at a dramatically lower price. Salesforce revolutionized CRM by offering their product for $10/user/month, while leading CRM solutions at that point were costing tens of thousands of dollars.

The key point here is that it is important to carve out a niche that you can dominate with your startup in order to get noticed in a growing industry and get your initial set of customers.

So, is this the end of my startup story?

No! In fact, this is just the beginning. Niche domination is not the aim. Industry domination is the aim. Visual Website Optimizer doesn’t only want to be the easiest A/B testing out there. In fact, it aims to be the leading A/B testing tool out there. It’s going to be hard — but not impossible. The idea is to expand feature-set horizontally and gain prominence outside of your niche slowly and steadily. Eventually, you should replace the industry leader and in fact become a source of market-driven ideas for other startups (I know, how meta).

Industry-leading companies are run by people similar to you, and they probably followed the path your startup is going to follow. So, there is no question that you can be an industry leader someday. It is a nice feeling to be a niche dominator, but don’t feel satisfied with it. Always set your eyes on the industry leadership position! That’s where the big bucks are. ■

Paras Chopra, based out of Delhi, India is the founder of Visual Website Optimizer, an A/B testing tool to help increase website sales and conversions. You can follow him on Twitter @paraschopra.

Reprinted with permission of the original author.
First appeared in *hn.my/money.*

The Lisp Curse

By RUDOLF WINESTOCK

The power of Lisp is its own worst enemy.

Here's a thought experiment to prove it: take two programming languages, neither of which are object-oriented. Your mission, if you choose to accept it, is to make them object-oriented, keeping them backward-compatible with the original languages, modulo some edge cases. Inserting any pair of programming languages into this thought experiment will show that this is easier with some languages than with others. That's the point of the thought experiment. Here's a trivial example: Intercal and Pascal.

Now make this thought experiment interesting: imagine adding object orientation to the C and Scheme programming languages. Making Scheme object-oriented is a sophomore homework assignment. On the other hand, adding object orientation to C requires the programming chops of Bjarne Stroustrup.

The consequence of this divergence in needed talent and effort causes *The Lisp Curse*:

Lisp is so powerful that problems which are technical issues in other programming languages are social issues in Lisp.

Consider the case of Scheme, again. Since making Scheme object-oriented is so easy, many Scheme hackers have done so. More to the point, many individual Scheme hackers have done so. In the 1990s, this led to a veritable warehouse inventory list of object-oriented packages for the language. *The Paradox of Choice*, alone, guaranteed that none of them would become standard. Now that some Scheme implementations have their own object orientation facilities, it's not so bad. Nevertheless, the fact that many of these packages were the work of lone individuals led to problems which Olin Shivers wrote about in documenting the Scheme Shell, scsh.

Programs written by individual hackers tend to follow the scratch-an-itch model. These programs will solve the problem that the hacker, himself, is having without necessarily handling related parts of the problem which would make the program more useful to others. Furthermore, the program is sure to work on that lone hacker's own setup, but may not be portable to other Scheme implementations or to the same Scheme implementation on other platforms. Documentation may be lacking. Being essentially a project done in the hacker's copious free time, the program is liable to suffer should real-life responsibilities intrude on the hacker. As Olin Shivers noted, this means that these 1-man-band projects tend to solve 80% of the problem.

Dr. Mark Tarver's essay, "The Bipolar Lisp Programmer," has an apt description of this phenomenon. He writes of these lone-wolf Lisp hackers and their

> *...inability to finish things off properly. The phrase "throw-away design" is absolutely made for the BBM, and it comes from the Lisp community. Lisp allows you to just chuck things off so easily, and it is easy to take this for granted. I saw this 10 years ago when looking for a GUI to my Lisp. No problem, there were 9 different offerings. The trouble was that none of the 9 were properly documented and none were bug free. Basically each person had implemented his own solution and it worked for him so that was fine. This is a BBM attitude; it works for me and I understand it. It is also the product of not needing or wanting anybody else's help to do something.*

ONCE AGAIN, CONSIDER the C programming language in that thought experiment. Due to the difficulty of making C object oriented, only two serious attempts at the problem have made any traction: C++ and Objective-C. Objective-C is most popular on the Macintosh, while C++ rules everywhere else. That means that, for a given platform, the question of which object-oriented extension of C to use has already been answered definitively. That means that the object-orientated facilities for those languages have been documented, that integrated development environments are aware of them, that code libraries are compatible with them, and so forth.

Dr. Mark Tarver's essay on bipolar Lispers makes the point:

> *Now in contrast, the C/C++ approach is quite different. It's so damn hard to do anything with tweezers and glue that anything significant you do will be a real achievement. You want to document it. Also you're liable to need help in any C project of significant size; so you're liable to be social and work with others. You need to, just to get somewhere.*
>
> *And all that, from the point of view of an employer, is attractive. Ten people who communicate, document things properly, and work together are preferable to one BBM hacking Lisp who can only be replaced by another BBM (if you can find one) in the not unlikely event that he will, at some time, go down without being rebootable.*

Therefore, those who already know C don't ask, "what object system should I learn?" Instead, they use C++ or Objective-C depending on what their colleagues are using, then move on to "how do I use object-oriented feature *X*?" Answer: "Goog it and ye shall find."

Real Hackers, of course, have long known that object-oriented programming is not the panacea that its partisans have claimed. Real Hackers have moved on to more advanced concepts such as immutable data structures, type inferencing, lazy evaluation, monads, arrows, pattern matching, constraint-based programming, and so forth. Real Hackers have also known, for a while, that C and C++ are not appropriate for most programs that don't need to do arbitrary bit-fiddling. Nevertheless, the Lisp Curse still holds.

Some smug Lisp-lovers have surveyed the current crop of academic languages (Haskell, Ocaml, et cetera) and found them wanting, saying that any feature of theirs is either already present in Lisp or can be easily implemented — and improved upon — with Lisp macros. They're probably right.

Pity the Lisp hackers.

Dr. Mark Tarver — twice-quoted, above — wrote a dialect of Lisp called Qi. It is less than 10,000 lines of macros running atop Clisp. It implements most of the unique features of Haskell and OCaml. In some respects, Qi surpasses them. For instance, Qi's type inferencing engine is *Turing complete*. In a world where teams of talented academics were needed to write Haskell, one man, Dr. Tarver wrote Qi all by his lonesome.

Read that paragraph again and extrapolate.

Exercise for the reader: Imagine that a strong rivalry develops between Haskell and Common Lisp. What happens next?

Answer: The Lisp Curse kicks in. Every second or third serious Lisp hacker will roll his own implementation of lazy evaluation, functional purity, arrows, pattern matching, type inferencing, and the rest. Most of these projects will be lone-wolf operations. Thus, they will have 80% of the features that most people need (a different 80% in each case). They will be poorly documented. They will not be portable across Lisp systems. Some will show great promise before being abandoned while the project maintainer goes off to pay his bills. Several will beat Haskell along this or that dimension (again, a different one in each case), but their acceptance will be hampered by flame wars on the comp.lang.lisp Usenet group.

Endgame: A random old-time Lisp hacker's collection of macros will add up to an undocumented, unportable, bug-ridden implementation of 80% of Haskell because Lisp is more powerful than Haskell.

The moral of this story is that **secondary and tertiary effects matter**. Technology not only affects what we can do with respect to technological issues, it also affects our social behavior. This social behavior can loop back and affect the original technological issues under consideration.

Lisp is a painfully eloquent exemplar of this lesson. Lisp is so powerful, that it encourages individual independence to the point of bloody-mindedness. This independence has produced stunningly good innovation as in the Lisp Machine days. This same independence also hampers efforts to revive the "Lisp all the way down" systems of old; no "Lisp OS" project has gathered critical mass since the demise of Symbolics and LMI.

One result of these secondary and tertiary effects is that, even if Lisp is the most expressive language ever, such that it is theoretically impossible to make a more expressive language, *Lispers will still have things to learn from other programming languages*. The Smalltalk guys taught everyone — including Lisp hackers — a thing or two about object oriented programming. The Clean programming language and the Mozart/Oz combo may have a few surprises of their own.

The Lisp Curse does not contradict the maxim of Stanislav Datskovskiy: **employers much prefer that workers be fungible, rather than maximally productive.** Too true. With great difficulty does anyone plumb the venality of the managerial class. However, the last lines of his essay are problematic. To wit:

As for the "free software" world, it eagerly opposes industrial dogmas in rhetoric but not at all in practice. No concept shunned by cube farm hells has ever gained real traction among the amateur masses.

In a footnote, he offers Linux as an example of this unwillingness to pursue different ideas. To be sure, he has a point when it comes to operating systems (the topmost comment, in particular, is infuriatingly obtuse). He does not have a point when it comes to programming languages. Python and Ruby were influenced by Lisp. Many of their fans express respect for Lisp and some of their interest has augmented the Lisp renaissance. With some justice, JavaScript has been described as "Scheme in C's clothing" despite originating in those cube farm hells.

Nevertheless, in spite of this influence, in both the corporate and open source worlds, Lisp still has only a fraction of the developer mind share which the current crop of advanced scripting languages have attracted. The closed-mindedness of MBA's cannot be the only explanation for this. The Lisp Curse has more explanatory power.

The free development environments available for Lisp further exemplify the Lisp Curse.

It's embarrassing to point this out, but it must be done. Forget about the Lisp Machine; we don't even have development systems that match what the average Smalltalk hacker takes for granted ("I've always felt Lisp is the superior language and Smalltalk is the superior environment," said Ramon Leon). Unless they pay thousands of dollars, Lisp hackers are still stuck with Emacs.

James Gosling, the author of the first Emacs that ran on Unix, has correctly pointed out that Emacs has not fundamentally changed in more than 20 years. This is because the Emacs maintainers are still layering code atop a design which was settled back when Emacs was a grad-student project at the MIT AI Lab, i.e., when Emacs development was still being indirectly financed by the national debt. A Slashdotter may object that Emacs is already quite capable and can do anything that any other development environment can do, only better. Those who have used Lisp Machines say otherwise.

So why don't the Lisp hackers put the Smalltalk guys in their proper place? Why don't they make a free development system that calls to mind some of the lost glories of the LispM, even if they can't reproduce another LispM?

The reason why this doesn't happen is because of the Lisp Curse. Large numbers of Lisp hackers would have to cooperate with each other. Look more closely: large numbers of the kind of people who become Lisp hackers would have to cooperate with each other. And they would have to cooperate with each other on a design which was not already a given from the beginning. And there wouldn't be any external discipline, such as a venture capitalist or other corporate master, to keep them on track.

Every project has friction between members, disagreements, conflicts over style and philosophy. These social problems are counteracted by the fact that no large project can be accomplished otherwise. "We must all hang together, or we will all hang separately." But the expressiveness of Lisp makes this countervailing force much weaker; one can always start one's own project. Thus, individual hackers decide that the trouble isn't worth it. So they either quit the project, or don't join the project to begin with. This is the Lisp Curse.

One could even hack Emacs to get something that's good enough. Thus, the Lisp Curse is the ally of Worse is Better.

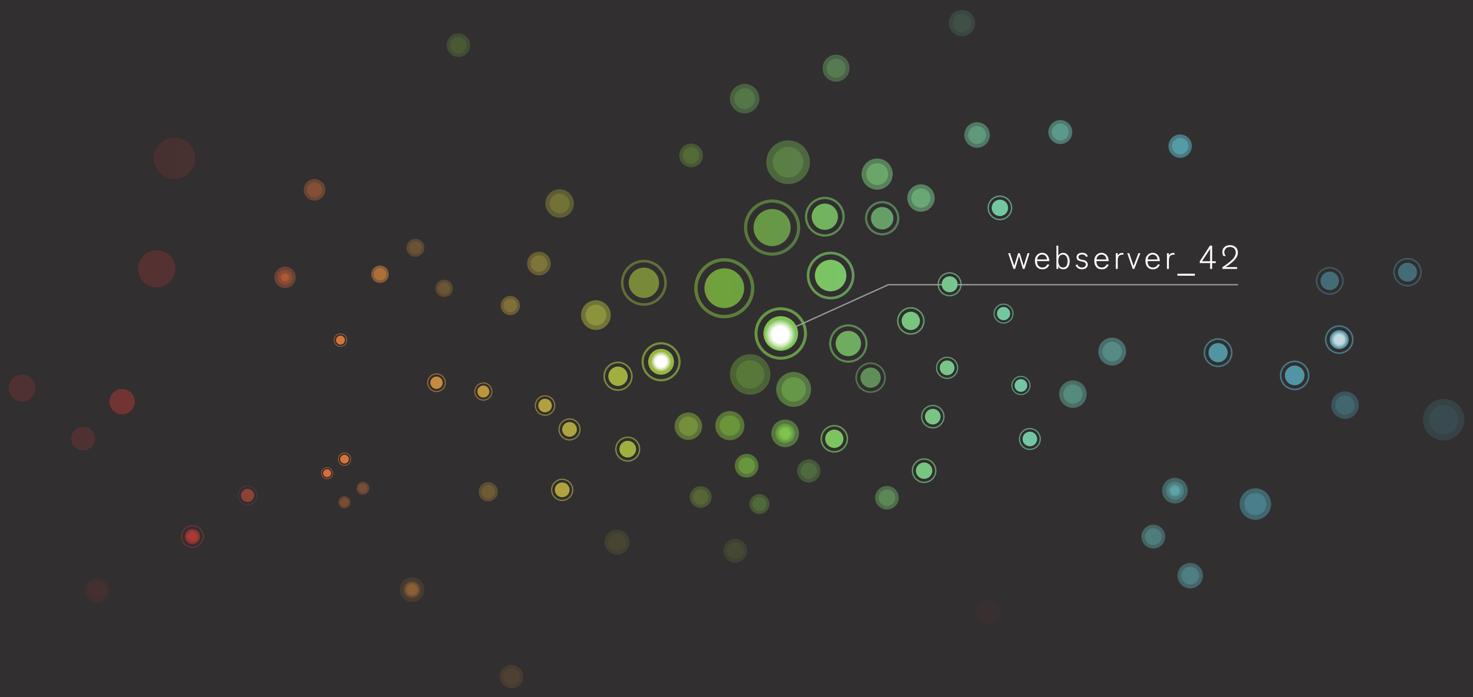
The expressive power of Lisp has drawbacks. There is no such thing as a free lunch. ■

Rudolf Winestock is an aspiring mathematician and writer with his own web design company at Winestock Webdesign, LLC.

These are your servers

● ● ●

These are your servers on Cloudkick



Any questions?

cloudkick.com
415.779.5425

support for 8 clouds + dedicated hardware

cloudkick
the best way to manage the cloud

Building a Web Application That Makes $500 a Month

By THOMAS BUCK

THIS IS AN article about the first web app I wrote for myself, TweetingMachine [tweetingmachine.com]. I'll cover every aspect of its creation and development, starting at how the idea came to me, the many, many mistakes I made, and how eventually I improved the tool so much that it now brings in $500 a month, a figure that increases with each month. I realize that this isn't a huge amount of money, but it's a nice present.

December 2009: The Idea

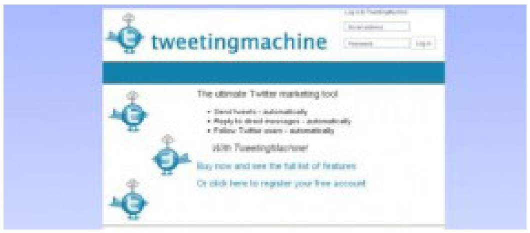
At the time, I was getting freelance work from vWorker [vworker.com], and I started to see a lot of requests asking for coders to work on various Twitter-based applications. Some people wanted to create sites that let users schedule tweets; others wanted to be able to automatically follow people back; and some shady characters wanted full-on spam engines. I was looking for an excuse to learn Twitter's API, and the more I thought about it, the more I realized that I could write a web app in my free time with lots of great features, that would be easy to use, and in no time it would become the #1 Twitter tool! Not only that — I could charge to access it…and people would sign up, and use it, and love it, and inside 90 days I'd be making tens of thousands each and every month!

Well, a guy can dream.

January 2010: The Execution

I had my great idea. Time to get cracking on what would turn out to be the easy bit: writing the code. I'm a web developer — have been for a decade — and I know how to write web apps. Find a cheap VPS (prgmr.com — incidentally, highly recommended and far exceeded my expectations), sketch out some database and object designs, choose a framework, and that was me, up and running, coding like a demon for a good few weeks.

The important fact here is that I'm very much a developer; I have all the design skills of a dead fish. So I took a look at a few sites out there and attempted to make something similar. This is going to be embarrassing, but here we go anyway:



The ultimate Twitter marketing tool

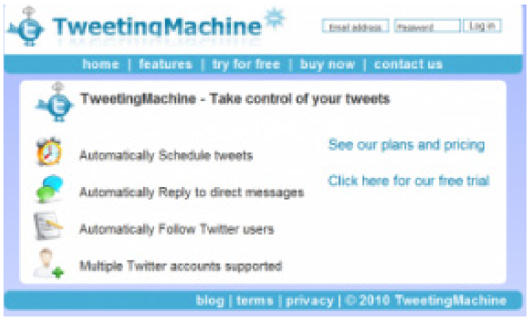
- Send tweets - automatically
- Reply to direct messages - automatically
- Follow Twitter users - automatically

With TweetingMachine!

Buy now and see the full list of features

Or click here to register your free account

First try



TweetingMachine
Email address
Password
Log in
home | features | try for free | buy now | contact us
See our plans and pricing
Click here for our free trial
blog | terms | privacy | © 2010 TweetingMachine

TweetingMachine - Take control of your tweets

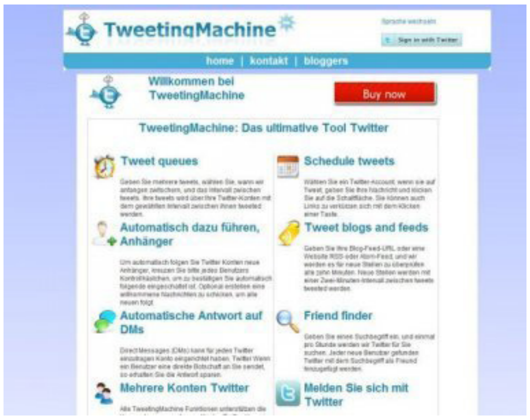
Automatically Schedule tweets

Automatically Reply to direct messages

Automatically Follow Twitter users

Multiple Twitter accounts supported

Let's try that again



Willkommen bei TweetingMachine

TweetingMachine: Das ultimative Tool Twitter

Tweet queues

Schedule tweets

Automatisch dazu führen, Anhänger

Tweet blogs and feeds

Automatische Antwort auf DMs

Friend finder

Mehrere Konten Twitter

Melden Sie sich mit Twitter

And again…

As you can see, TweetingMachine was not a pretty sight. I was still naively optimistic that my poor design skills would be ignored by the legions of customers that would be overawed by TweetingMachine's features and ease of use. I launched the site, submitted it to the likes of FeedMyApp, KillerStartups, and so on. This was right before…

February 2010: The Big Pause

My girlfriend and I (along with her sister, for that matter) had decided to escape Poland's chilly winter and spend three weeks in India instead. A fantastic time was had by all, and I occasionally managed to stop thinking about the millions of dollars that must be waiting in my PayPal account.

March 2010: Crashing Back Down to Reality

Arrive back home. Check emails. Zero sales. Check server. Apache has been crashing. Cron jobs not running. Sit down. Cry. Fix up the code. Go work on something else.

April 2010: First Sale!

I should stop here to explain what I originally thought my pricing plans would look like: I was offering tiered pricing — if you wanted to use multiple Twitter accounts, it'd cost you more…and if you wanted to send more messages, that would cost you as well. Enjoying taking rash decisions, I decided to scrap the tiered pricing, and stick to a single price: $9.99/month, with a week's free trial beforehand.

Surprisingly, within a week, I had my first sale. With $9.99 in my PayPal account, I was halfway to breaking even on my monthly hosting costs, a small triumph! That said, I was starting to notice a rather nasty trend: my visitor numbers were dropping, sharply. If this carried on, I would have maybe a single visitor per day in the next month. Not having any marketing skills, I was starting to wonder what I should do.

“I got into an argument — “It’s only $19.99, why don’t you just buy it?!””

May 2010: Internet Marketing for Dummies

I was at a loss, and so I started to read every basic guide out there for how to market your web app. All of them made it seem so simple: find relevant websites and blogs; contact authors and owners; ask for a review or if they’d let you publish something; and then sit back and watch the targeted visitors pour in.

Sadly, with TweetingMachine that didn’t happen. I started to realize that its design could really be holding the tool back, but I don’t have the money to pay a designer, so what else can I do? Failing elsewhere, I added a page to the site “Bloggers” that offered a free year’s subscription to TweetingMachine in return for a review on their blog.

Just in case you ever go down this route, you will not believe the cheek of some people. I still regularly receive emails from people demanding free subscriptions, and sending me a link to a copy of a review by someone else. Funnily enough though, in a couple of cases this has led to purchases after I got into an argument — “It’s only $19.99, why don’t you just buy it?!” — with the person originally trying to cheat a subscription out of me.

June 2010: Second Sale, and Desperation Kicks In

Suddenly, my second sale arrived: I was now breaking even on my monthly hosting costs! I decided to ignore the design problem: with enough features, SEO, and gimmicks, surely I’d start to make enough money to pay for a designer? So, in my free time I worked on these 3 aspects:

❶ **As mentioned, adding more features.** Otherwise known as reading my competitor’s websites, and working out how to do what they’re doing, but do it better.

❷ **SEO.** I started reading every SEO guide out there after realizing how many basic mistakes I was making (such as having a title tag consisting of the word TweetingMachine alone)

❸ **Gimmicks.** Another embarrassing confession, but honestly, this is how desperate I was. I made TweetingMachine translation-friendly, and then set about adding Google Translate versions of every language I could find. I later realized quite how terrible and irritating the translations were when the visitor logs showed non-English visitors repeatedly choosing the English version of the site, usually after viewing a single page in their native tongue.

“Decent designs are available for not much money at all!”

July 2010 – September 2010: Close to Giving Up

The pattern of low usage and sales continued over the next few months. I gained 10 subscribers, over half of whom cancelled after a month's usage. And, honestly, I lost interest in the project, now hating the design and the feature set.

One evening, though, I got in contact with a friend I hadn't spoken to in ages. He mentioned a website that was paying his rent, and I expressed my frustration about TweetingMachine's lack of income. I think at this point, total monthly revenue was $30.

Have you ever felt really, really stupid? I excel in stupidity, missing common sense and so on, and as the conversation progressed, the familiar feeling swept over me once again. My friend told me, “Yup, honestly your site's design sucks. Why don't you go on ThemeForest, buy this theme [hn.my/agencia] for the front-end, this theme [hn.my/terminator] for the tool itself, and hey — you only need a couple of subscribers for a couple of months, and the themes will have paid for themselves.”

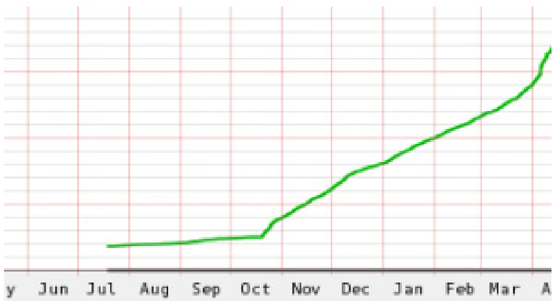
Well, knock me down with a feather! Decent designs are available for not much money at all! My friend had made a great argument. I paid the $50 and got to work.

October 2010: What a Difference a Design Makes

I bought the themes and sat down to integrate them. I was expecting this to take a lot longer than it did: in the end, it took me a few hours over the course of the evening. Bedtime was approaching, and I chose to spend the last hour of the night harassing my ever-patient fiancée with over-enthusiastic demonstrations of TweetingMachine's new-found greatness.

You see, over the past few months, my hatred for TweetingMachine had built up day by day, its cheery colors and shiny logo only heightening my sense of failure. Thankfully, integrating the themes gave me a new burst of enthusiasm for the project. Suddenly, I was really enjoying visiting the site and playing with the tool. As I woke up the next day, my head filled with all-too-ambitious dreams of wealth and success, and this in turn motivated me to develop yet more features.

Enough talking! What do the figures look like? This is a graph of new free trials:



I'll leave the precise date I put the new designs live as an exercise for the reader.

So, cue wild happiness! But how well did this translate into sales?

Honestly, I couldn't believe it.

One month, and only five new subscriptions.

November 2010: Running Out of Excuses

At this point, I was charging $9.99/month, with a free trial of 24 hours. I thought this was a fair deal, but as the stagnating number of new subscriptions showed, it was clear that something wasn't working.

What could it be? And why? Of course! The price! That must be where I'm going wrong!

Halfway through November, I bit the bullet and made a significant change to the pricing: TweetingMachine now cost $19.99 per year; no more monthly rebills.

I have yet to have any magic moments with changes I've made to TweetingMachine. By that, I mean, changes that have had an instant effect on use or payment. The pricing change was no exception…for the first 2 days.

It was now the middle of November. And on the 15th, someone subscribed. On the 16th, another user paid. Teasingly, no-one signed up on the 17th…but on the 18th I received 2 new subscriptions. The pattern continued for the rest of the month, averaging 1 sale per day.

Well! This was much more like it! November brought in over $200!

December 2010: What Was that About Marketing?

The rate of new subscriptions continued throughout December, sometimes 2 subscriptions a day, and on 1 memorable day, 5 users subscribed!

So now the concept is proven, how can I get more potential users to visit the site? I took the view that if in doubt (as I was and continue to be — note earlier implied reference to business ability of a squashed frog) follow what your competitors are doing. So, I sat down, typed my competitors' names into Google, went through page after page of links, and identified bloggers who might be interested in covering TweetingMachine. I sent hundreds of personalized emails, and received fewer than 10 reviews in total. Oh well, better than a kick in the teeth.

I also found lots of directories of Twitter tools: type in a description, upload screenshots, get listed, lots of happy users dance their merry way towards your site.

I was expecting new subscriptions to tail off just before Christmas. It made sense to me that there might not be that many people online, and even fewer ready to hand over their hard-earned cash after the yuletide spending craziness. Imagine my happiness when new subscriptions continued, including on Christmas day itself!

December brought in just under $500 — to be precise, $479.71 after PayPal fees.

January 2011: Brave New Subscriptions

The first month of 2011 was a time of high emotion. For several days on end, no users would sign up…and then a flurry of 3 or 4 subscriptions would come in within a couple of hours. My mind even deluded itself into thinking it was acceptable to describe this as "A rollercoaster of emotions."

Whilst I was happy with January's takings — $429.75 after fees — the new signups permanently tailed off at the end of the month. Something was definitely amiss.

February 2011: Delusions

I had started a new job in January, developing Facebook applications for a local startup, and this was taking up an awful lot of my evening time. I had some ideas for changes to make to TweetingMachine that I was keen to implement, but wasn't sure when I'd get around to it. After all, assuming that my time is free, each month it still bought in the equivalent of a few nice meals out.

Still, subscriptions had nearly stopped coming in altogether. For reasons unknown, TweetingMachine had 6 people in total subscribing in February. I had strangely depressing thought: if I had been lucky in December and January, how much money is being made by those who knew what they were doing?

I eventually found the spare time, and made a couple of small, but effective changes: I coded some flexibility into prices, so future price changes would take seconds to implement, and I did the same for the free trial period.

Going forwards, TweetingMachine cost $19.99/month, and the free trial increased from 24 hours to 10 days.

March 2011: All Change!

What difference did increasing the price 12-fold make? Color me shocked, surprised and, frankly, happy: it made *zero* difference!

Actually, I tell a lie; the rate was between that of January and February. Essentially the same — except that these subscriptions were going to be rebilled each and every month!

I was starting to feel cautiously optimistic. There were still plenty of outstanding questions (such as: how do I get more, more, more users to visit?), but for now my most pressing question was: will these subscribers continue their subscriptions next month?

April 2011: The answer is… yes!

Please forgive me if this article feels like a con. I've had months where I've made just under $500, and months where I've made a lot less than that. April's on target to make over $500, and as of the 27th, I've had 1 unsubscribe from the previous month, out of 17 new subscribers. Not the greatest retention rate, but if that continues for the next 20 years…

Now, this isn't a story of huge, wild success; it's of a 29-year-old making his first steps in business. I believe that in a few more years, after a few more failures, and a few more, modest, successes, I'll be in a pretty good place for my first major success. The point is: you have to make a start. ■

Thomas is a British developer currently based in Warsaw, Poland. When he's not offending tunefully-gifted listeners with his woeful piano skills, he blogs about his attempts to create profitable SaaS apps at *tbbuck.com*.

Reprinted with permission of the original author.
First appeared in *hn.my/500*.

Hiring Developers: You're Doing It Wrong

By UDO SCHROETER

When Evan Carmi posted his Google job interview experience [hn.my/ecarmi] on HN, I felt reminded of my bygone startup days. In over a decade of "modern" IT startup job interviews, we have made no progress whatsoever. If anything, I was part of the problem there for a few years. I simply copied a hiring mechanism that seemed like a standard at the time, and in doing so I failed miserably at the most important goals a company should observe when looking for new developers. Today the tech front pages are full of Larry Page's efforts to turn around the company, but I think performance problems at developer-centric companies may to a large part be burned into their DNA by a deeply faulty hiring process.

How We Did It

My cofounder and I were running a small web development shop in Germany. We had started working literally out of my friend's basement. Over time, we grew and moved into real office space. At first it was easy to find new employees, we could just ask our friends to come in and work for us. Of course, that model didn't scale, but it performed a very important function: it made sure we hired people that were a good fit for the company, both on a personal and a professional level. Then came the day when we needed to fill positions by bringing in people from the outside.

One of the redeeming features of the German regional unemployment offices is they will send you a large stack of CVs on demand, within a few hours of calling them on the phone. I was pleasantly surprised that we didn't have to hire an agency to do this. Together with the CVs we already had from people who applied to the job posting on our website, we now had some sifting to do. In the end, we agreed on about a dozen of the best and invited them for an interview. This is the part where everything went wrong.

The Standard Dev Interview

A candidate would come in, usually all dressed up in their best suit and tie, and we'd sit down and have a talk. That talk was essentially like an oral exam in college. I would ask them to code algorithms for all the usual cute little CS problems, and I'd get answers with wildly varying qualities. Some were shooting their pre-canned answers at me with unreasonable speed. They were prepared for exactly this kind of interview. Others would break under the "pressure", barely able to continue the interview.

To be honest, when we first started doing this, I had to look up these puzzles in advance, mainly to make sure I didn't embarrass myself. This should have been the first warning sign that maybe we weren't exactly testing for skills that were most relevant to our requirements. If these doubts occurred to me, I must have dismissed them very quickly. After all, it was the way everyone approached the interview process.

Of course, we ended up hiring the candidate with the smoothest answers. Inevitably, the next job openings came, we did it again and again in the same fashion, for the rest of the company's lifetime. If this sounds familiar to you, you are clearly not alone.

Actual Job Performance

But how did the candidates we selected measure up? The truth is, we got very mixed results. Many of them were average, very few were excellent, and some were absolutely awful fits for their positions. So at best, the interview had no actual effect on the quality of people we were selecting, and I'm afraid that at worst, we may have skewed the scale in favor of the bad ones.

What does bad and good even mean in this context? Let's have a look some of the benchmarks that I consider important:

Company Culture. In hindsight, one of the most important features a new employee should have is compatibility with the spirit of the people who already work there. The Standard Dev Interview performed worst in this area, for obvious reasons. It's difficult to judge people's personalities in interviews because they are not exactly themselves. In fact, they're incentivized not to be themselves.

Programming Competence. Somewhat counter intuitively, the code examples done during the interview were a bad indicator for actual competence on the job. Real world projects rarely consist of implementing binary searches without access to a parser or literature. It turned out that employees who had done very well in the code examples were not always able to transfer theoretical knowledge into practical solutions. Having candidates write sorting algos on the whiteboard is a method that rewards people with great and precise short-term memory who come prepared for exactly these kinds of questions. In our case, we needed resourceful coders who could write neat, stable, and elegant software — and the interview process wasn't delivering them to us.

Project Management. People who did well in the interview were not necessarily good team mates or even good presenters in front of our customers. This result, too, was surprising to me. Turns out, sucking up to an interviewer for an hour is a completely different skill set than, say, being good at coordinating with your coworkers or the people who pay our bills. Nor was their interview performance indicative of the ability to write good documentation or how to behave in online communications.

The Result

The results of a hiring process such as this may be one of the factors responsible for a company to lose its startup spirit and its creative soul. This was certainly the case with our company. As the CEO, I was ultimately at fault; however, having the wrong people on the job was a large part of the company's inability to deliver the quality and quantity of output needed to sustain it. Infighting poisoned our teams. Incompetence was covered up with good presentation skills and ass-kissing. Good people left the company because they hated the new atmosphere.

Though I had to let go of many people for different reasons over the years and in the end had to deliver the hardest speech of my life on the morning I dissolved the company, I only went "full Trump" once on an employee. It was the one who had displayed the best interview performance and the best academic references of them all, only a year before.

Sure, that's an extreme example. Most companies succeed regardless. But I still believe we can vastly improve the chances of finding candidates that are good fits by radically changing the way we do interviews. And in our case, that would probably have made all the difference in the world.

An Alternative

So what should a developer job interview look like then? Simple: eliminate the exam part of the interview altogether. Instead, ask a few open-ended questions that invite your candidates to elaborate about their programming work.

- What's the last project you worked on at your former employer?
- Tell me about some of your favorite projects.
- What projects are you working on in your spare time?
- What online hacker communities do you participate in?
- Tell me about some (programming/technical) issues that you feel passionately about.

These questions are designed to reveal a great deal about the person you have in front of you. They can help you decide whether the candidate is interested in the same things as you, whether you like their way of thinking, and where their real interests lie. It's tougher for them to bullshit their way through here, because the interviewer can drill deeper into a large number of issues as they present themselves.

What about actual coding ability? Well, take a few moments after the interview and look into some code the candidate wrote. Maybe for an open source project, maybe they have to send you something that's not public — it doesn't matter. Looking at actual production code tells you so much more than having them write contrived fiveliners on the whiteboard.

I'm sure you can come up with even more questions and other ways to engage the interviewee. At this point, pretty much any idea will be an improvement.

Nay-saying

Most people are quick to defend the status quo, and sure, that's a rewarding position to hold. It's risk free and you can always fall back on the old argument "a lot of smart, rich and successful people do it the old way, so my money is on whatever they are doing."

"Nice, but that doesn't work for large, successful companies. Your idea doesn't scale."
Sure, it scales: in terms of effort per interview, there is no difference. There is no reason this couldn't work in larger companies. In the end, the interviewer always makes a personal and deeply subjective decision. I'm merely suggesting a way that delivers more relevant information for that purpose.

"The best programmers have no spare time projects." or: "The most talented people I know work from 9 to 5 and then go home to watch football/be with their families/whatever."
This is not my experience. I'm not saying that a good programmer should not have a life. But I do believe that a certain amount of enthusiasm for programming is called for. And really, if you have such a great skill, not using it for fun seems kind of wasteful to me.

"In my spare time I'm working on making the next million for my company. Oh, when I'm not working for my company? I'm with my family or friends."
That's great, those people can surely show me something they have been working on. I would, however, consider the lack of any hobby projects a warning sign for some development jobs.

Final Thoughts

It has been my experience that the traditional developer interview is insufficient at finding good candidates. While the typical whiteboard coding exercises correlate somewhat with general CS competence, they are poor indicators of actual programming performance. It is my contention that we have been doing them this way for years simply because they're easy to administer, but the data that's coming out of these interviews is largely irrelevant at best. We as an industry should move to more personalized interview questions that focus on the entirety of a developer's skill set. Also, I believe it is more productive to judge production code as opposed to abstract modular puzzles that have no real connection to the actual job in question. Most importantly, I am convinced that gaining insight into the developer's real personality is just as important as checking for professional competence, because one bad fit can destroy an entire team. ■

Udo Schroeter works as a project manager at Kautschuk Gesellschaft Group in Frankfurt, Germany. While he loves writing web applications, his professional focus is bioinformatics and computational modeling. In his spare time, he is currently building the Hubbub Distributed Social Network open source project.

Reprinted with permission of the original author.
First appeared in *hn.my/hiring*.

A Rough Guide to Social Skills for Awkward Smart People

By KENNETH MYERS

I AM A FULL-ON dork. The things that make me want to get up in the morning are things that make normal people lose interest in the conversation, or giggle. These are things like lucid dreaming, artificial intelligence, utopian movements, and Esperanto.

Be that as it may, I'm mostly fine with boring the normals and living in the Vibrant True World of Beauty with its other full-on dork denizens. Amazingly, I've found that Esperantists seem to be anarcho-Taoists, that AI researchers tend to have experimented with lucid dreaming, and that other secret threads hold the seemingly disparate interests of Dorks Like Me together. I have countrymen. Just not yet my country.

The other thing that holds my kinsmen together, though, is an unfortunate thing: they are all asses. They decimate the chances of their ideas' success by offending everyone they meet, making it look like being happy and having friends are suspicious, counterrevolutionary behaviors.

In case you're wondering if my sermon is directed at you, there are some common tropes in our oft-reenacted social suicide:

❶ We call someone's beliefs "idiotic."

❷ We call someone's beliefs "idiotic" within five minutes of meeting them.

❸ We happily inform strangers of our vast and superior intelligence.

❹ We derail a conversation about American Idol to bring it back to the real issue at hand: that there is no God.

❺ When given a compliment, "Oh, you're so well-read!", we look blankly in the eyes of the complimenter, and respond "Yes, I know."

I can hear your retort, oh ye smart and lonely: "But I am the smartest person in the room"/"But their beliefs are idiotic"/"I'm not going to compromise the truth to make some idiot happy."

Great. Good luck with that. Oh, and by the way, your cause will die, I promise.

People don't respond well to being told that they're idiots, even if they are. Ideas don't spread by beating their enemies to a pulp. They spread by subterfuge and incalculable subtlety.

I would propose that sacrificing some smaller truths in your day-to-day interactions is the only way for the greater truth to prevail.

Be a Good Spy

As a short exercise, I invite you to think of it this way: it is World War II, and you are an Allied spy. You are in Germany, and you have attained a mid-level rank in the Nazi bureaucracy. Your superiors speak well of the Führer.

Now ask yourself, which response probably achieves the most towards the furtherance of your objectives?

a) "No, he's actually an idiot, and killing Jews is wrong, and I'm an Allied spy, and there are Jews in my attic."

or

b) "Heil Hitler."

The Old One-Two

Now, of course we'll never achieve anything good if we simply walk around saying "Heil Hitler" all day. If you do have an important mission in the world, you'll have to face dangers, and at some point show your true colors.

Doing this in the wrong way Schrutes your whole mission. Doing this in the right way makes you Ani Difranco or Bob Dylan.

Ani Difranco has a trick. She gets up on the stage, and her guitar is un-tuned. While tuning it, she ad-libs a story. The story isn't funny. There are a lot of pauses, and a lot of "uh"s. The crowd starts to get uncomfortable. We feel sympathetic embarrassment. Massive pity. Poor little girl. Then, suddenly, she rips into everyone's soul, fast. Now she's confident and smarter than you can handle. Now she's referencing poets and playing brilliantly with language. The whole dumb scared thing was an act (she doesn't do it in interviews). It works. I call this The Old One-Two.

One: Disarm. Don't be an ass. Be weak. Be self-deprecating. Build Ethos.

Two: Be brilliant.

The Old One-Two is charm at its atomistic simplest. Most good actors use it (though not so much in their stage performances as in interviews). Bob Dylan is the absolute king of the game, ripping off Milton and making it sound like something he misheard his grandfather say.

What I find the most interesting about The Old One-Two is that even after I realize I've been duped, I still love the guy who's scammed me.

"Oh no, I really don't play piano, I just mess around."

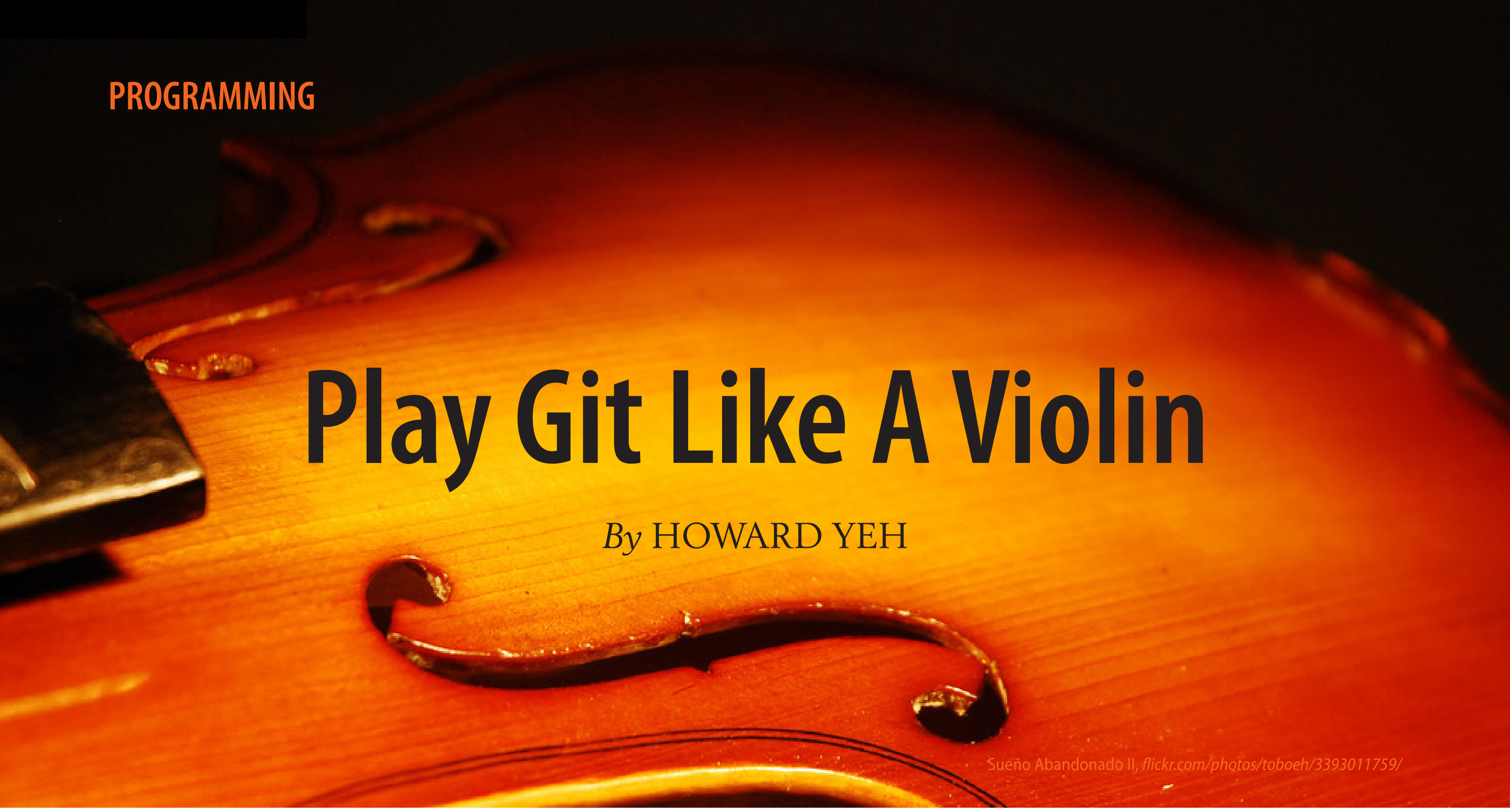
"Aw, come on, please?"

"Oh, alright." {Flawless Bach Piece}

"Whoa."

Even after you know it was a lie, the false humility still gives you warm feelings. Now when this guy later turns around and says, "Aw, naw, not really — well, I guess kind of I dabble in The Ultimate Truth," I'll probably listen. ■

Kenneth Myers is the administrator of an ESL program at a small college in Texas, an amateur programmer, an occasional politician, and a fun guy. Call him when you're in Dallas. You should be his friend.



Play Git Like A Violin

By HOWARD YEH

People think that playing the violin is hard. But that's only when you are learning and practicing. When you are actually playing, it's as natural as breathing. So it is with Git. After a couple years of use, and with the help of a few aliases, my Git usage now comes as easily as music from a familiar piece:

```
git caa
git ca
git s
git l
git r1
git rh 330183
git s
git d
git cm 'a new commit'
```

All of us at some point or another kept a private cheat sheet of common Git commands. I know I did. After a while, I gained enough experience with Git to know the common tasks that I do all the time. For these common tasks I create aliases.

Very often, after I've made a commit, I'd keep wanting to make small fixes to it, like fussing around with spaces, or renaming variables, or rewording the comments, or minor refactoring of the code. These changes are too small to be worth their own commits (that would only clutter up the history). So I'd prefer if these changes belonged to the commit I already have. I'd do this:

```
git commit -a --amend -C HEAD
```

This adds all the changes to the staging area and commits it as an amendment to the previous commit, using the same commit message. Effectively, I am saying: "put whatever I've done into the previous commit."

For this usage pattern, I have created an alias in my `~/.gitconfig`, like so:

```
[alias]
  caa = commit -a --amend -C HEAD
```

Then, ever after, I'd type `git caa` whenever I wanted to do the same thing. Another pattern I use a lot is to create a commit for the changes I've done, all in one step:

```
git commit -a -m 'commit message'
```

Thus I'd create another alias:

```
[alias]
  cma = commit -a -m
```

Then, ever after that, I'd type `g cma`. Ninety percent of the time, `git caa` and `git cma` cover my commit needs. If you ask me what they stand for, I honestly can't tell you, because these commands are so short, they are ingrained in my muscle memory. I don't think about what I am doing with Git, just as when I am playing an arpeggio on the violin, I don't think about the notes individually.

Here are all my Git aliases. I hope you find some of them useful to integrate into your Git workflow.

```
[alias]
# I like using the interactive mode to
# make complex commits
ai = add --interactive

# All the aliases relate to commits. Note
# that they are grouped by common prefixes,
# so I don't confuse what I want done by
# accident.

c = commit
# commit with a message
cm = commit -m
cma = commit -a -m
# amending the previous commit
ca = commit --amend
caa = commit -a --amend -C HEAD
# reset
## soft resets
r = reset
r1 = reset HEAD^
r2 = reset HEAD^^
## hard resets
rh = reset --hard
rh1 = reset HEAD^ --hard
rh1 = reset HEAD^^ --hard

# shortcuts for commands
s = status
d = diff
a = add
co = checkout
b = branch
l = log
f = fetch
r = reset
p = push
```

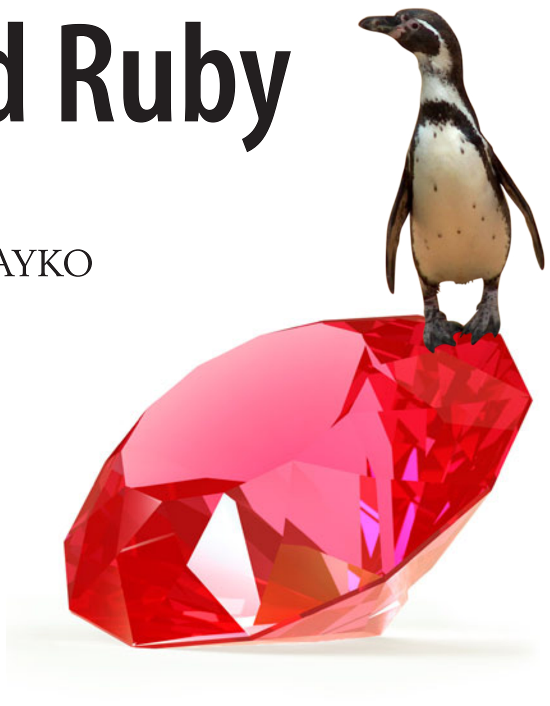
Cherry on top: I aliased `g` as `git` in my bash shell. What I actually do is:

```
g caa
g ca
g s
g l
g r1
g rh 330183
g s
g d
g cm 'a new commit'
```
■

Howard Yeh graduated with a degree in Cognitive Science and currently travelling the world. He likes Lisp, but now works mostly in Ruby. Follow him on Twitter @hayeah.

AWK-ward Ruby

By RYAN TOMAYKO



RUBY, LIKE MOST successful languages, was assembled from pieces of things that came before it: Smalltalk's consistent object system, Perl's practical syntax, UNIX's sensibilities. Not that it didn't bring entirely new innovations of its own(Smalltalk had block syntax first!), but it's amazing to consider how much of Ruby's design rests on the elegant packaging of old concepts into a new coherent whole.

There's something less obvious but perhaps more essential that Ruby borrowed: the very concept of blatant, unashamed borrowing. In his 1999 talk, *Perl, the first postmodern computer language*, Larry Wall states plainly that Perl was built mostly from things that "didn't suck" in the languages that preceded it:

> *When I started designing Perl, I explicitly set out to deconstruct all the computer languages I knew and recombine or reconstruct them in a different way, because there were many things I liked about other languages, and many things I disliked. I lovingly reused features from many languages. (I suppose a Modernist would say I stole the features, since Modernists are hung up about originality.) Whatever the verb you choose, I've done it over the course of the years from C, sh, csh, grep, sed, awk, Fortran, COBOL, PL/I, BASIC-PLUS, SNOBOL, Lisp, Ada, C++, and Python. To name a few. To the extent that Perl rules rather than sucks, it's because the various features of these languages ruled rather than sucked.*

Ruby, the story goes, borrowed much from Perl: integral regular expressions, statement modifiers (`do_this if that`), array/hash literals, funny global variable names, and, of course, the philosophy of having more than one way of doing the same thing (TMTOWTDI).

Or did it?

If these features didn't originate with Perl, as Wall seems to imply, then where did they come from?

Image of Ruby licensed by DepositPhotos.com/Kesamasek

One of the most important influences on Perl's design was AWK. So much so that Perl was sometimes described as a semantic superset of AWK. Are the relics of AWK still present in Ruby? Let's see.

Today, AWK is probably best known as a versatile tool for extracting fields from delimited flat files in a shell pipeline:

```
cat /etc/passwd | awk -F: '{ print $1 }'
```

It's rare to see AWK used for more complex problems in modern systems, but there's actually a full-blown programming language lurking beneath the surface. It was at one time used to solve a lot of the same problems people commonly use Ruby, Perl, or Python to solve today.

You might find some of AWK's language features familiar:

- Associative array type.
- Automatic string, integer, and floating point value types.
- C-style `if`, `while`, and `do` constructs.
- For-each style `for` construct for iterating over associative arrays.
- Arithmetic (`+`, `-`, `*`, `/`), modulu-division (`%`), exponentiation (`^`), increment/decrement (`++`, `--`), and assignment shorthand (`+=`, `-=`, `*=`, …) operators.
- Array membership operator (`expr in array`).
- Integral regular expression type and matching operators (`str ~ /pattern/`).
- Comprehensive builtin function library (a small sample: `printf`, `gsub`, `split`, `substr`, `cos`, `sin`, `log`, `sqrt`).
- User defined functions.

Not bad for 1977.

It would seem that a large portion of Ruby's basic syntax and semantics were present in AWK. So how did Perl come to dominate the problem space? There must be something very different about AWK.

While AWK had much of the primitive syntax right, it also overcompensated for a specific case: processing streams of delimited text. The top-level context is used exclusively for declaring one or more matching statements:

```
pattern { action }
...
```

Here, `pattern` is a full-blown expression and `action` is a block of code executed when pattern evaluates truthfully. The `pattern` is tested for each line (or record) of input and `action` is executed when `pattern` returns truthfully. Omitting the `pattern` causes the action to be executed for every line.

There's special patterns for setting actions up to run before the first line of input is read and after all lines have been processed. Here's an example that uses the special `BEGIN` pattern along with a regular expression match. It prints all the usernames from `/etc/passwd` while avoiding comment lines:

```
cat /etc/passwd |
awk '
  BEGIN     { FS = ":" }
  /^[a-z_]/ { print $1 }
'
```

(NOTE: You can paste bomb that into your shell on just about any UNIX system.)

Pingouin, *flickr.com/photos/52345210@N08/4816336371/*

Here's a more complex example that shows off some of AWK's advanced features, like associative arrays and for-in syntax. It calculates word frequencies from the text of Jonathan Swift's, *A Modest Proposal*:

```
curl -s http://www.gutenberg.org/
files/1080/1080.txt |
awk '
  BEGIN { FS="[^a-zA-Z]+" }

  {
    for (i=1; i<=NF; i++) {
      word = tolower($i)
      words[word]++
    }
  }

  END {
    for (w in words)
      printf("%3d %s\n", words[w], w)
  }
' |
sort -rn
```

It may seem strange, but this style of programming was very common in UNIX's hayday. Instead of programs being dominated by a single language like Perl or Ruby, you'd build pipelines that combined standard utilities (like `sort` shown above), sprinkle in bits and pieces of AWK as needed, and drop down to C when performance was critical.

Perl took the guts of AWK and left behind the mandatory pattern matching at the top-level. That simple design change turned what was a special purpose language for processing delimited text streams into what we know today as a "general purpose scripting language."

But that's not the end of the story.

It was important that Perl be able to act as a replacement for AWK in all its capacities, including within shell pipelines. This meant having the ability to run `perl` in a kind of top-level AWK mode. Ruby borrowed this capability from Perl, making it possible to use Ruby for the same style of programming facilitated by AWK, complete with `BEGIN` and `END` blocks!

Here's the word frequency script in AWK-ish Ruby:

```
curl -s http://www.gutenberg.org/
files/1080/1080.txt |
ruby -ne '
  BEGIN { $words = Hash.new(0) }
  $_.split(/[^a-zA-Z]+/).each
  { |word| $words[word.downcase] += 1 }

  END {
    $words.each { |word, i|
    printf "%3d %s\n", i, word }
  }
' |
sort -rn
```

The `-n` argument causes Ruby to assume a `while gets(); ... end` loop around the provided script. `$_` is set to the last line read, and the `BEGIN` and `END` blocks function exactly as they did in AWK. ■

Ryan Tomayko is a systems designer at GitHub and lifelong student of Unix philosophy.

DKIM, SPF, Mime Parsing,
Postfix, Throttling, Feedback loops…

This is what it takes to get to the inbox.
Let us deal with it.

Postmark

hacker.postmarkapp.com
Transactional email delivery for web apps.

Understanding JavaScript's "this" Keyword

By ANGUS CROLL

The JavaScript this keyword is ubiquitous, yet misconceptions abound.

What You Need to Know

Every execution context has an associated `ThisBinding` whose lifespan is equal to that of the execution context and whose value is constant. There are 3 types of execution context:

❶ Global context

`this` is bound to the global object (`window` in a browser)

```
alert(this); //window
```

❷ Function context

There are at least 5 ways to invoke a function. The value of `this` depends on the method of invocation.

a) Invoke a property

`this` is the `baseValue` of the property reference:

```
var a = {
    b: function() {
        return this;
    }
};

a.b(); //a;

a['b'](); //a;

var c= {};
c.d = a.b;
c.d(); //c
```

b) Invoke a variable

this is the global object:

```
var a = {
    b: function() {
        return this;
    }
};

var foo = a.b;
foo(); //window

var a = {
    b: function() {
        var c = function() {
            return this;
        };
        return c();
    }
};

a.b(); //window
```

The same applies to self-invoking functions:

```
var a = {
    b: function() {
        return this;
    }
};

var foo = a.b;
foo(); //window

var a = {
    b: function() {
        var c = function() {
            return this;
        };
        return c();
    }
};
a.b(); //window
```

c) Invoke using Function.prototype.call

this is passed by argument.

d) Invoke using Function.prototype.apply

this is passed by argument:

```
var a = {
    b: function() {
        return this;
    }
};

var d = {};

a.b.apply(d); //d
```

e) Invoke a constructor using new

this is the newly created object:

```
var A = function() {
    this.toString = function() {
        return "I'm an A"
    };
};

new A(); //"I'm an A"
```

❸ Evaluation context

this value is taken from the this value of the calling execution context:

```
alert(eval('this==window'));
//true - (except firebug, see above)

var a = {
    b: function() {
        eval('alert(this==a)');
    }
};

a.b(); //true;
```

What You Might Want to Know

This section explores the process by which `this` gets its value in the functional context — using ECMA 5 262 as a reference.

Let's start with the ECMAScript definition of `this`:

> The `this` keyword evaluates to the value of the `ThisBinding` of the current execution context.
>
> *– from ECMA 5, 11.1.1*

How is *ThisBinding* set?

Each function defines a [[Call]] internal method which passes invocation values to the function's execution context:

> The following steps are performed when control enters the execution context for function code contained in function object F, a caller provided `thisValue`, and a caller provided `argumentsList`:
>
> 1. If the function code is strict code, set the `ThisBinding` to `thisValue`.
>
> 2. Else if `thisValue` is null or undefined, set the `ThisBinding` to the global object.
>
> 3. Else if `Type(thisValue)` is not `Object`, set the `ThisBinding` to `ToObject(thisValue)`.
>
> 4. Else set the `ThisBinding` to `thisValue`.
>
> *– from ECMA 5, 10.4.3 Entering Function Code (slightly edited)*

In other words, `ThisBinding` is set to the object coercion of the abstract argument `thisValue`, or if `thisValue` is undefined, the global object (unless running in strict mode, in which case, `thisValue` is assigned to `ThisBinding` as-is).

So where does thisValue come from?

Here we need to go back to our 5 types of function invocation:

❶ Invoke a property
❷ Invoke a variable

In ECMAScript parlance these are *Function Calls* and have two components: a *MemberExpression* and an *Arguments* list.

> 1. Let `ref` be the result of evaluating *MemberExpression*.
>
> 2. Let `func` be `GetValue(ref)`.
>
> 6. If `Type(ref)` is `Reference`, then
>
> a. If `IsPropertyReference(ref)` is true
>
> i. Let `thisValue` be `GetBase(ref)`.
>
> b. Else, the base of `ref` is an Environment Record
>
> i. Let `thisValue` be the result of calling the `ImplicitThisValue` concrete method of `GetBase(ref)`.
>
> 8. Return the result of calling the [[Call]] internal method on `func`, providing `thisValue` as the `this` value and providing the list `argList` as the argument values
>
> *– from ECMA 5, 11.2.3 Function Calls*

So, in essence, `thisValue` becomes the `baseValue` of the function expression (see step 6, above).

Where the function is expressed as a **property**, the `baseValue` is the identifier preceding the dot (or square bracket).

foo.bar(); //`foo` assigned to `thisValue`
foo['bar'](); //`foo` assigned to `thisValue`

```
var foo = {
    bar:function() {
        //(Comments apply to example
        //invocation only)
        //MemberExpression = foo.bar
        //thisValue = foo
        //ThisBinding = foo
        return this;
    }
};
foo.bar(); //foo
```

For **variables**, the `baseValue` is the VariableObject (the "Environment Record" above), which is a *Declarative Environment Record*. ECMA 10.2.1.1 tells us that the `ImplcitThisValue` of a Declarative Environment Record is undefined.

var bar = function() {…};
bar(); //`thisValue` is undefined

Revisiting 10.4.3 Entering Function Code (see above) we see that unless in strict mode, an undefined `thisValue` results in a `ThisBinding` value of global object. So `this` in a variable function invocation will be the global object.

In full…

```
var bar = function() {
    //(Comments apply to example
    //invocation only)
    //MemberExpression = bar
    //thisValue = undefined
    //ThisBinding = global object
    //(e.g.window)
    return this
};
bar(); //window
```

❸ Invoke using Function.prototype.apply

❹ Invoke using Function.prototype.call

(specifications at **15.3.4.3 Function.prototype.apply** and **15.3.4.4 Function.prototype.call**)

These sections describe how, in `call` and `apply` invocations, the actual value of the function's `this` argument (i.e. its first argument) is passed as the `thisValue` to 10.4.3 Entering Function Code. (Note this differs from ECMA 3, where primitive `thisArg` values undergo a `toObject` transformation, and null or undefined values are converted to the global object — but the difference will normally be negligible since the value will undergo identical transformations in the target function invocation [as we've already seen in 10.4.3 Entering Function Code.])

❺ Invoke a constructor using new

> When the [[Construct]] internal method for a Function object F is called with a possibly empty list of arguments, the following steps are taken:
>
> 1. Let `obj` be a newly created native ECMAScript object.
>
> 8. Let result be the result of calling the [[Call]] internal property of F, providing `obj` as the `thisValue` and providing the argument list passed into [[Construct]] as args.
>
> 10. Return `obj`.
>
> *– from ECMA 5, 13.2.2 [[Construct]]*

This is pretty clear. Invoking the constructor with `new` creates an object that gets assigned as the `thisValue`. It's also a radical departure from any other usage of `this`.

House Cleaning

Strict mode

In ECMAScript's strict mode, the `thisValue` is not coerced to an object. A `this` value of `null` or `undefined` is not converted to the global object and primitive values are not converted to wrapper objects.

The bind function

`Function.prototype.bind` is new in ECMAScript 5 but will already be familiar to users of major frameworks. Based on call/apply, it allows you to prebake the `thisValue` of an execution context using simple syntax. This is especially useful for event handling code, for example, a function to be invoked by a button click, where the `ThisBinding` of the handler will default to the `baseValue` of the property being invoked — i.e. the button element:

```
//Bad Example: fails because ThisBinding
//of handler will be button
var sorter = {
    sort: function() {
        alert('sorting');
    },
    requestSorting: function() {
        this.sort();
    }
}
$('sortButton').onclick = sorter.
requestSorting;
```

```
//Good Example: sorter baked into This
//Binding of handler
var sorter = {
    sort: function() {
        alert('sorting');
    },
    requestSorting: function() {
        this.sort();
    }
}
$('sortButton').onclick = sorter.request-
Sorting.bind(sorter);
```

Further Reading

ECMA 262 5th Edition (PDF) [hn.my/emca]

- 11.1.1 Definition of this
- 10.4.3 Entering Function Code
- 11.2.3 Function Calls
- 13.2.1 [[Call]]
- 10.2.1.1 Declarative Environment Record (ImplicitThisValue)
- 13.2.2 [[Construct]]
- 15.3.4.3 Function.prototype.apply
- 15.3.4.4 Function.prototype.call
- 15.3.4.5 Function.prototype.bind
- Annex C The Strict Mode of ECMAScript ■

Angus Croll is a front end developer at Twitter and author of the influential "JavaScript, JavaScript" blog [javascriptweblog.wordpress.com]. He's also a mentor at *JSMentors.com*.

Parsing: The Solved Problem That Isn't

By LAURENCE TRATT

PARSING IS THE act of taking a stream of characters and deducing if and how they conform to an underlying grammar. For example the sentence, "Bill hits Ben," conforms to the part of the English grammar `noun-verb-noun`. Parsing concerns itself with uncovering structure; although this gives a partial indication of the meaning of a sentence, the full meaning is only uncovered by later stages of processing. Parsable but obviously nonsensical sentences, like "Bill evaporates Ben," highlight this (the sentence is still `noun-verb-noun`, but finding two people who agree on what it means will be a struggle). As humans we naturally parse text all the time, without even thinking about it; indeed, we even have a fairly good ability to parse constructs that we've never seen before.

In computing, parsing is also common. While the grammars are synthetic (e.g. of a specific programming language), the overall idea is the same as for human languages. Although different communities have different approaches to the practicalities of parsing — (C programmers reach for `lex/yacc`; functional programmers to parser combinators; others for tools like ANTLR or a Packrat/PEG-based approach), they typically rely on the same underlying area of knowledge.

After the creation of programming languages themselves, parsing was one of the first major areas tackled by theoretical computer science and, in many people's eyes, one of its greatest successes. The 1960s saw a concerted effort to uncover good theories and algorithms for parsing. Parsing in the early days seems to have shot off in many directions before (largely) converging. Context Free Grammars (CFGs) eventually won, because they are fairly expressive and easy to reason about, both for practitioners and theorists.

Unfortunately, given the extremely limited hardware of 1960s computers (not helped by the lack of an efficient algorithm), the parsing of an arbitrary CFG was too slow to be practical. Parsing algorithms such as LL, LR, and LALR identified subsets of the full class of CFGs that could be efficiently parsed. Later, relatively practical algorithms for parsing any CFG appeared, most notably Earley's 1973 parsing algorithm. It is easy to overlook the relative difference in performance between then and now: the *fastest computer in the world* from 1964-1969 was the CDC6600 which executed at around 10 MIPS. My 2010 mobile phone has a processor which runs at over 2000 MIPS. By the time computers had become fast enough for Earley's algorithm, LL, LR, and friends had established a cultural dominance which is only now being seriously challenged. Many of the most widely used tools still use those algorithms (or variants) for parsing. Nevertheless in tools such as ACCENT / ENTIRE and recent versions of `bison`, one has access to performant parsers that can parse any CFG, if that is needed.

The general consensus, therefore, is that parsing is a solved problem. If you've got a parsing problem for synthetic languages, one of the existing tools should do the job. A few heroic people — such as Terence Parr, Adrian Johnstone, and Elizabeth Scott — continue working away to ensure that parsing becomes even more efficient, but, ultimately, this will be transparently adopted by tools without overtly changing the way that parsing is typically done.

Language composition

One thing that's become increasingly obvious to me over the past few years is that the general consensus breaks down for one vital emerging trend: language composition. Composition is one of those long, complicated, but often vague terms that crops up a lot in theoretical work. Fortunately, for our purposes it means something simple: grammar composition, which is where we add one grammar to another and have the combined grammar parse text in the new language (exactly the sort of thing we want to do with Domain Specific Languages [DSLs]). To use a classic example, imagine that we wish to extend a Java-like language with SQL so that we can directly write:

```
for (String s : SELECT name FROM person
WHERE age > 18) {
  ...
}
```

Let's assume that someone has provided us with two separate grammars: one for the Java-like language and one for SQL. Grammar composition seems like it should be fairly easy. In practice, it turns out to be rather frustrating, and I'll now explain some of the reasons why.

Grammar composition
While grammar composition is theoretically trivial, simply squashing two grammars together is rarely useful in practice. Typically, grammars have a single start rule; one therefore needs to choose which of the two grammars has the start rule. More messy is the fact that the chances of the two grammars referencing each other is slight. In practice, one needs to specify a third tranche of data — often referred to, perhaps slightly misleadingly, as glue — which actually links the two grammars together. In our running example, the Java-like language has the main grammar; the glue will specify where, within the Java-like expressions, SQL statements can be referenced.

For those using old parsing algorithms such as LR (and LL etc.), there is a more fundamental problem. If one takes two LR-compatible grammars and combines them, the resulting grammar is not guaranteed to be LR-compatible (i.e. an LR parser may not be able to parse using it). Therefore such algorithms are of little use for grammar composition.

At this point, users of algorithms such as Earley's have a rather smugger look on their face. Since we know from grammar theory that unioning two CFGs always leads to a valid CFG, such algorithms can always parse the result of grammar composition. But, perhaps inevitably, there are problems.

Tokenization
Parsing is generally a two-phase process: first we break the input up into tokens (tokenization,; and then we parse the tokens. Tokens are what we call words in everyday language. In English, words are easily defined (roughly, a word starts and ends with a space or punctuation character). Different computer languages, however, have rather different notions of what their tokens are. Sometimes, tokenization rules are easily combined; however, since tokenization is done in ignorance of how the token will later be used, sometimes it is difficult. For example, in SQL, SELECT might be a keyword, but in Java it is also a valid identifier; it is often hard — if not impossible — to combine such tokenization rules in traditional parsers.

Fortunately, there is a solution: scannerless parsing (e.g. SDF2 scannerless parsing). For our purposes, it might perhaps better be called tokenless parsing. The different names reflect the naming conventions of different parsing schools. Scannerless parsing does away with a separate tokenization phase. The grammar now contains the information necessary to dynamically tokenize text. Combining grammars with markedly different tokenization rules is now possible.

Fine-grained composition

In practice, the simple "glue" mentioned earlier used to combine two grammars is often not enough. There can be subtle conflicts between the grammars, in the sense that the combined language might not give the result that was expected. Consider combining two grammars that have different keywords. Scannerless parsing allows us to combine the two grammars, but we may wish to ensure that the combined languages do not allow users to use keywords in the other language as identifiers. There is no easy way to express this in normal CFGs. The SDF2 paper referenced earlier allows reject productions as a solution to this; unfortunately this then makes SDF2 grammars mildly context sensitive. As far as I know, the precise consequences of this haven't been explored, but it does mean that at least some of the body of CFG theory won't be applicable; it's enough to make one a little nervous, at the very least (not withstanding the excellent work that has been created using the SDF2 formalism by Eeclo Visser and others).

A recent, albeit relatively unknown, alternative are boolean grammars. These are a generalization of CFGs that include conjunction and negation, which, at first glance, are exactly the constructs needed to make grammar composition practical (allowing one to say things like "identifiers are any sequence of ASCII characters except SELECT"). Boolean grammars, to me at least, seem to have a lot of promise, and Alexander Okhotin is making an heroic effort on them. However, there hasn't yet been any practical use of them that I know of, so wrapping one's head around the practicalities is far from trivial. There are also several open questions about Boolean grammars, some of which, until they are answered one way or the other, may preclude wide-scale uptake. In particular, one issue relates to ambiguity, of which more now needs to be said.

Ambiguity

By severely restricting which CFGs they accept, grammars that are compatible with traditional parsing algorithms (LL, LR etc.) are always unambiguous (though, as we shall see, this does not mean that all the incompatible grammars are ambiguous-many are unambiguous). Grammar ambiguity is thus less widely understood than it might otherwise have been. Consider the following grammar of standard arithmetic:

```
E ::= E "+" E
    | E "-" E
    | E "/" E
    | E "*" E
```

Using this grammar, a string such as `2 + 3 * 4` can be parsed ambiguously in two ways: as equivalent to `(2 + 3) * 4;` or as equivalent to `2 + (3 * 4)`. Parsing algorithms such as Earley's will generate all possibilities even though we often only want one of them (due to arithmetic conventions, in this case we want the latter parse). There are several different ways of disambiguating grammars, such as precedences (in this example, higher precedences win in the face of ambiguity):

```
E ::= E "+" E  %precedence 1
    | E "-" E  %precedence 1
    | E "/" E  %precedence 2
    | E "*" E  %precedence 3
```

This might suggest that we can tame ambiguity relatively easily. Unfortunately, parsing theory tells us that the reality is rather tricky. The basic issue is that, in general, we cannot statically analyze a CFG and determine if it is ambiguous or not. To discover whether a given CFG is ambiguous or not, we have to try every possible input: if no input triggers an ambiguous parse, the CFG is not ambiguous. However, this is, in general, impractical: most CFGs describe infinite languages and cannot be exhaustively tested. There are various techniques that aim to give good heuristics for ambiguity (see Bas Basten's masters thesis [hn.my/basten] for a good summary; I am also collaborating with a colleague on a new approach, though it's far too early to say if it will be useful or not). However, these heuristics are inherently limited. If they say a CFG is ambiguous, it definitely is; but if they cannot find ambiguity, all they can say is that the CFG *might* be unambiguous.

Since theoretical problems are not always practical ones, a good question is the following: is this a real problem? In my experience thus far, defining stand-alone grammars for programming languages using Earley parsing (i.e. a parsing algorithm in which ambiguity is possible), it has not been a huge problem. As the grammar designer, I often understand where dangerous ambiguity might exist and can nip it in the bud. I've been caught out a couple of times, but not enough to really worry about.

However, I do not think that my experience will hold in the face of widespread grammar composition. The theoretical reason is easily stated: combining two unambiguous grammars may result in an ambiguous grammar (which, as previously stated, we are unlikely to be able to statically determine in general). Consider combining two grammars from different authors, neither of whom could have anticipated the particular composition: it seems to me that ambiguity is much more likely to crop up in such cases. It will then remain undetected until an unfortunate user finds an input that triggers the ambiguity. Compilers that fail on seemingly valid input are unlikely to be popular.

PEGs

As stated earlier, unambiguous parsing algorithms such as LL and LR aren't easily usable in grammar composition. More recently, a rediscovered parsing approach has gathered a lot of attention: Packrat/PEG parsing (which I henceforth refer to as PEGs). PEGs are different than everything mentioned previously: they have no formal relation to CFGs. The chief reason for this is PEGs *ordered choice* operator, which removes any possibility for ambiguity in PEGs. PEGs are interesting because, unlike LL and LR, they're closed under composition: in other words, if you have two PEGs and compose them, you have a valid PEG.

Are PEGs the answer to our problems? Alas — at least as things stand now — I doubt it. First, PEGs are rather inexpressive: like LL and LR parsing, PEGs are often frustrating to use in practise. This is, principally, because they don't support left recursion. Alex Warth proposed an approach which adds left recursion, but I discovered what appear to be problems with it, though I should note that there is not yet a general consensus on this (and I am collaborating with a colleague to try and reach an understanding of precisely what left recursion in PEGs should mean). Second, while PEGs are always unambiguous, depending on the glue one uses during composition, the ordered choice operator may cause strings that were previously accepted in the individual languages not to be accepted in the combined language — which, to put it mildly, is unlikely to be the desired behaviour.

Conclusions

If you've got this far, well done. This article has ended up much longer than I originally expected — though far shorter than it could be if I really went into detail on some of these points! It is important to note that **I am not a parsing expert**: I only ever wanted to be a *user* of parsing, not — as I currently am — someone who knows bits and pieces about its inner workings. What's happened is that, in wanting to make greater use of parsing, I have gradually become aware of the limitations of what I have been able to find. The emphasis is on "gradually": knowledge about parsing is scattered over several decades (from the 60s right up to the present day), from many publications (some of them hard to get hold of) and many people's heads (some of whom no longer work in computing, let alone in the area of parsing). It is therefore hard to get an understanding of the range of approaches or their limitations. This article is my attempt to write down my current understanding and, in particular, the limitations of current approaches when composing grammars. I welcome corrections from those more knowledgeable than myself. Predicting the future is a mugs game, but I am starting to wonder whether, if we fail to come up with more suitable parsing algorithms, programming languages of the future that wish to allow syntax extension will bypass parsing altogether, and use syntax-directed editing instead. Many people think parsing is a solved problem — I think it isn't. ■

Dr. Laurence Tratt is a software consultant and Senior Lecturer at Middlesex University. His research interests center around language design, implementation, and usage. His homepage can be found at *tratt.net/laurie*.

The business book for entrepreneurs: over 30 interviews with startup founders.

Follow along as a Y Combinator alumni interviews other startup founders to understand how they deal with challenges such as finding cofounders, raising money, getting users, staying motivated, and hiring.

STARTUPS
OPEN SOURCED

NEW

stories to inspire & educate

Grooveshark Reddit GitHub Foursquare Airbnb Weebly Greplin AppSumo Wufoo LittleAppFactory MixPanel LikeALittle Djangy Divvyshot Justin.TV Blippy Bump WePay DailyBooth Gobble KISSMetrics Omnisio Cloudkick Noteleaf OneLlama Octopart Crowdbooster Listia Hipmunk Indinero OrangeQC Camino Real One

Jared Tame

What people are saying

"This is great! Having interviewed over 300 founders, I'm impressed - if you want to understand how some of today's most impressive startup founders got to where they are, this book will deliver."
- Andrew Warner, Mixergy.com

"One of the richest collections of authentic and inspiring stories."
- Avichal Garg, BluePrint Labs cofounder

"I think this book will end up causing more students to try to become entrepreneurs."
- Niniane Wang, Cofounder Sunfire Offices

Special discount for Hacker Monthly readers!

First 250 May edition readers receive 15% off at **StartupsOpenSourced.com**, just use coupon code "**hmonthly511**"

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.

Please contact promo@magcloud.com with any questions.

MAGCLOUD