Sam Altman

The Days are Long But the Decades are Short

HACKERMONTHLY

Issue 61 June 2015

Curator
Lim Cheng Soon

Contributors
Heidi Rozen
Sam Altman
Andrew Montalenti
Keegan McAllister
Greg Baugues
James Rowe
Evan Miller
Matthew Griffith

Proofreader
Emily Griffin

Illustrator
Alla Berlezova

Printer
Blurb

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

Advertising
ads@hackermonthly.com

Contact
contact@hackermonthly.com

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

Contents

FEATURES

PROGRAMMING

For links to Hacker News dicussions, visit *hackermonthly.com/issue-61*

Illustration by: Alla Berlezova [nightmarev.com]

How to Build a Unicorn From Scratch — and Walk Away with Nothing

By HEIDI ROIZEN

This is a grim fairy tale about a mythical company and its mythical founder. While I concocted this story, I did so by drawing upon my sixteen years of experience as a venture capitalist, plus the fourteen years I spent before that as an entrepreneur. I'm going to use some pretty simple math and some pretty basic terms to create a really awful situation in the hopes that entrepreneurs reading this might avoid doing the same in the real world.

As I've seen over many years and many deals, in all but the most glorious outcomes, terms will matter way more than valuations, and way more than whatever your cap table says. And yet entrepreneurs — often with the encouragement of their stakeholders — optimize for the wrong things when they negotiate their financings.

This is my attempt to paint you a picture of why this is such a bad idea. The situation I present is fake, but the outcome is remarkably similar to those I've witnessed. Don't let this happen to you.

Let's start with our entrepreneur, whom we'll call Richard. He's founded a breakthrough company. Let's call it Pied Piper.

Richard attracts Peter, a newly-wealthy budding angel investor, who agrees to put in $1 million as a note with a $5 million cap and a 20% discount.

With his $1 million, Richard builds a small team of people, rents an Eichler in Palo Alto, and gets to work. Once he is able to demonstrate his product, he heads to Sand Hill Road. He's in a hot space in a hot market. He nails his pitch, and the term sheets roll in.

Because Richard is extremely sensitive to dilution (after all, he's seen The Social Network) he wants the highest valuation possible. (Early in my career, another venture capitalist called valuation "the grade at the top of the paper" — and I've never forgotten that.) The highest valuation, $40 million pre-money, comes from an emerging venture fund, let's call them BreakThroughVest (BTV). BTV is excited about this deal, but has 'ownership requirements' of at least 20%, so they insist that to support that valuation they need to invest $10 million. Plus, they want a senior liquidity preference of 1x to protect their downside since they feel the valuation is rich given the stage of the company.

Richard is thrilled with the valuation and the fresh capital for only 20% dilution. The prior investor, Peter, is stoked that he is getting his $1 million investment converted into roughly 20% of this super hot company, and now with the validation of an external term sheet he can mark his position up to $10 million, a 10X! This helps Peter validate his position as a savvy angel and solidify his syndicate following on AngelList.

Term sheet signed. Champagne popped. A few weeks later, funds wired.

With the $10 million, Richard rents space in SoMa on a seven-year lease, hires lots more people, and within a few months he is able to roll out the minimally viable product to test the market. Awash in the buzz of his fundraise, a feature in Re/code, and some early user traction, Pied Piper is perceived as the emerging leader in a nascent, winner-take-all market. While they are not yet monetizing their users, the adoption metrics are off the charts.

Pied Piper attracts the attention of a tech giant we'll just call Hooli. Hooli's consumer group wants access to Pied Piper's data. With Hooli dollars behind Pied Piper, Pied Piper could inundate the market with consumer facing advertising to build their user base and upend competitors given the massive network effect of the product. Hooli approaches Richard with the idea of a large strategic round. In the deal, Hooli would invest $200 million for equity while in return the two companies would enter into a business development agreement on the side in which Pied Piper guarantees to spend that money in a massive consumer campaign on Hooli's ad platform. They float the magic "B" valuation. Richard goes to sleep dreaming of rainbows and unicorns.

Richard fantasizes about being named a member of the Unicorn Club by the press. His employees calculate the huge paper gains on their options — they will all be instant millionaires — and since no one is more than ¼ vested, they are all highly motivated to stay in spite of long, long work hours. BTV is thrilled with the 20x markup on Pied Piper, since they are about to hit their LPs up for a new fund. The original investor, Peter, has achieved legendary status — his $1 million has turned into approximately $200 million on paper. He's on the YC VIP sneak preview list, he's been offered a spot on Shark Tank, and Ashton just called to try to get into his next deal.

Of course, that $200 million for 20% stake also comes in with a senior 1x liquidation preference in order for Hooli to create sufficient downside protection and thereby justify the $1 billion valuation to their board.

Richard, Peter and BTV all agree it is worth doing. With $200 million to spend on the most massive consumer-facing ad campaign in this sector's history, the $1 billion valuation will seem low in retrospect.

Except, it doesn't end up happening that way.

The ads start running, but the conversion rate is low. Pied Piper shows Hooli the atrocious metrics and demands out of the advertising commitment, but Hooli won't budge: Performance metrics were not pre-negotiated, and furthermore the ad group that recommended the investment did so in part to prop up their revenues with Pied Piper's money 'round-tripping' into their coffers. The ad group is counting on that money to hit their annual numbers.

Pied Piper is forced to run the whole campaign, blowing through all $200 million. The good news: They increased their user base by 10x. The bad news: The resulting business model those users end up actually supporting equates to more of a 'market valuation' of $200 million. In more bad news, turns out Richard incorrectly estimated the cost of supporting those users, most of whom are taking advantage of the 'free' part of a freemium model. Support costs skyrocket.

Word about the poor conversion leaks out. The advertising stops when the money runs out. Growth slows to a trickle when the advertising stops. New investors sniff around, but with the preference overhang of $211 million, they are concerned about employees being buried under that structure and therefore being unmotivated to continue. They ask prior investors to recap, but the investors don't want to give up their preferences: Pied Piper is now looking like it might be worth far less than the paper valuation, which means those preferences are very valuable as downside protection. Furthermore, BTV is out raising their fund, and the last thing they want to do is write down their 10x markup on the Pied Piper investment.

The board is now super unhappy about the massive miscalculation of support costs, awful user conversion, gargantuan ad overspend, the lack of growth the company is experiencing, and the departure of a few key employees who've seen this movie before and have done the 'overhang math.' Richard as CEO is out of his element — the problems are huge and the company needs more money, which he is incapable of raising given his lack of experience navigating waters like these. Unfortunately, it is the CEO's job to fix problems and raise money, and if he can't do it, someone else has to. So the board (which now controls the company with 60% of the stock) votes to remove Richard as CEO. They recruit an interim CEO (let's call him George) to quickly take the helm. George says he'll take the job on two conditions: One, that they create a 5% carve-out for him and the go-forward employees (he's done the overhang math, too) and two, that they extend the runway so he has time to either turn this thing around — or sell it.

The company is not profitable and the current investors are tapped out. "Let's extend the runway using debt," says BTV. Maybe things will improve with time — or at least perhaps they can get their fund closed before they have to take the write down.

They lean on their good friends at PierLast Venture Bank who cough up $15 million in debt, with a senior preference and a 2x guarantee. Onerous terms to be sure, but hard to get debt with a balance sheet like this. Unfortunately, Pied Piper is burning $2 million a month on office space, cloud services, customer support, and expensive employees who are needed to build the next generation of the product. Without support they'd have to shut down existing customers and revenue, yet without development of the new release that they hope will save the company, they will have nothing to sell. Since they can't cut their way to glory, they have to simply hope they can grow into their valuation.

Time ticks by while the company plods forward with very slow growth. Market pressures force them to lower prices, pushing profitability off. A few key developers leave. Once again, they are facing the prospect of running out of money in 90 days. Current investors are worried. Not only do they not have funds to put into the deal, but once payroll is missed they could be personally liable for the damage. Not good.

Luckily, WhiteKnight, a public company with a complementary product and plenty of cash, offers to buy Pied Piper. The offer is $250 million. It's not a billion — but it's still a big, impressive number. It's not that easy to create a company worth a quarter billion real dollars to someone else. That's huge!

The venture debt provider PierLast is very nervous about Pied Piper's balance sheet and looks to the VCs to either guarantee the loan or get the sale done. They want their $30 million. Hooli is likewise pushing to sell, after all they are guaranteed the first $200 million of any proceeds, after repayment of 2x debt to PierLast, while the company would have to be worth over a billion for them to see any further upside given that they only own 20%. Their calculus is that this is about as unlikely as seeing a real unicorn given the state of the company. BTV, who no longer has any capital left to invest from their original fund, has recently closed their shiny new $300 million fund, so they decide it is time to take their chips off the table. They vote to sell too, getting their $10 million back. Peter, while sad about the outcome, has developed a huge syndication following on AngelList and has recently benefitted from an early acquisition that netted him $3 million on a $250k investment. Can't win them all, but he's at peace. Even Richard votes yes to the sale: He still has a board seat but given the company's lack of profitability and lack of any other sources of capital, turning down this deal would mean insolvency, missed payroll, and personal liability. George (the interim CEO) and the key go-forward employees demand their $12.5 million carve-out. Tack on more money for lawyers and ibankers, and…

Oh wait, that's more than $250 million. Oops.

Ergo, Richard ends up with nothing.

So what can we learn from Richard's grim fairy tale?

Terms matter

Liquidation preferences, participation, ratchets — even the very term preferred shares (they are called 'preferred' for a reason) are things every entrepreneur needs to understand. Most terms are there because venture capitalists have created them, and they have created them because over time they have learned that terms are valuable ways to recover capital in downside outcomes and improve their share of the returns in moderate outcomes — which more than half the deals they do in normal markets will turn out to be.

There is nothing inherently evil about terms, they are a negotiation and part of standard procedure for high risk investing. But, for you the entrepreneur to be surprised after the fact about what the terms entitle the venture firm to is just bad business — on your part.

Cap tables don't tell the real story

For any private company with different classes of stock, the capitalization table is not-at-all the full picture of who gets what in an outcome.

In the above example, each of the three investors held 20% of the stock and Richard and crew held 40%, yet the outcome was vastly different because of those aforementioned pesky terms and preferences.

Before you close on any round, you should create a waterfall spreadsheet that shows what you and each other stakeholder would get in a range of exits — low, medium and high. What you will generally find is that, in high, everyone is happy. In low, no one is happy, and in medium (which is where most deals settle) you can either be penniless or "life-changingly" compensated, depending on how much money you raised and what terms you agreed to. It is simply foolish to sell part of the company you founded without understanding this fully.

This is why it is so crazy to me that many entrepreneurs today are focused on valuation — the grade at the top of the paper. They are willingly trading terms for a high number. Before you do so, run the math on the range of outcomes over multiple term and valuation scenarios, so you fully understand the tradeoffs you are making.

Venture capital is not free money. It's debt. And then some

People mistakenly think of an equity investment as 'only' equity dilution. After all, if you lose everything, your venture investor can't come after you for your house like a bank lender could. However, most all venture transactions are done for preferred shares with a liquidation preference, which means all that venture money is guaranteed to be paid back first out of any proceeds before you get to make a dime. The more money you raise, the higher that 'overhang' becomes. And interestingly, the higher the valuation, the higher the delta of value you need to create before the investor would rather hold on to the end instead of getting his or her money back (or a multiple thereof, as some terms dictate) in a premature sale if things are looking iffy. And what company doesn't go through iffy times?

Stacked preferences can create massive problems down the line

This one is a hard to articulate in a blog post. Plus, I am a venture capitalist who on occasion puts said senior preferences in my term sheet. They exist for a reason — again often to do with the valuation and the risk/reward tradeoff the investor needs to make using the downside protection of a senior preference against the minimization of dilution the entrepreneur wants to achieve with a sky high valuation. They are not inherently bad.

But regardless of why they are there, the more diversity of value and terms in each round, the more you will create a situation where your investors (who are almost always also your voting board members) will have very different return profiles on the same offer. In the above example (and again I apologize for simplified math but it is directionally accurate) Hooli is getting their $200 million back on a $250 million acquisition. They own only 20% because of the high valuation they paid. So for them to instead double their return, the company would have to go public for $2 billion! This is a case of the bird in the hand being worth more than the two in the very distant bush.

Investors are portfolio managers: You are not

You are betting usually 10 years of your life and all your available assets on your startup. Your investor is likely investing out of a fund where he or she will have 20-30 other positions. So in the simplest of terms, the outcome matters more to you than it does to them. As I noted above, when you have stacked preferences, each person at the table may be facing a vastly different outcome. But now layer onto that their fund or partner dynamics. Ever heard the expression, "lose the battle but win the war?" I've seen behavior that would seem crazy, until one considers what is going on in the background. For example in the above, BTV is out raising a fund and depends on that 10X markup to validate their abilities as investors. Facing a write down, a fire sale — or an extension of runway using debt (and not incurring any accounting change) — which one do you think least impacts the most important thing they are doing right now? For our angel Peter, whose star has risen with this legendary markup, what value is there to him of taking a $1 million loss right now instead of just leaving a walking dead company out there and on his books (although this company is not technically walking dead because, since it is not profitable, it is not walking. But I digress.)

Most reputable investors do not engage in this sort of optics, and many of us who have been through the dot com bust are actually rather aggressive with our write downs to accurately reflect a sense of true value in our portfolios. Also, most investors who are also board members wear multiple hats and take their fiduciary responsibilities very seriously — I know I do. But, I bring up these behaviors because I've witnessed them more than once out there in the real world. As an entrepreneur, you should at least think through the motivations of others, both when you are structuring investments as well as when you are considering a sale. They will on occasion matter… a lot.

What to do

Now that I've scared you, let me reiterate that most investors I deal with are great, ethical people. If I didn't think of venture capital money as good for entrepreneurs on the whole, I wouldn't be a venture capitalist. But we VCs do a lot more deals than you entrepreneurs do, and you need to go into them with your eyes open to the downside consequences of the terms you agree to.

Here's what I recommend:

- **Focus on terms, not just valuation**: Understand how they work. Read this book. [hn.my/vdeals] Use a lawyer that does tech venture financings for a living, not your uncle who is a divorce attorney, so you are getting the best advice. Don't

completely delegate this because you need to understand it yourself.

- **Build a waterfall**: Once you understand the terms being offered, build a waterfall spreadsheet so you can see exactly how each stakeholder will fare across the range of potential exit values (yes by stakeholder, not by class of stock: Investors often end up owning multiple classes, and likewise different people in the same class may have very different circumstances that will influence their behavior even in the same outcome.)
- **Don't do bad business deals just to get investment capital**: I know, *duh*, *right?* But I've seen otherwise brilliant entrepreneurs get entranced by these big number deals with big corporates, only to deeply regret them later when they cannot be unwound. My advice, separate the business development contract from the equity contract. Negotiate them individually. If the business development deal would not stand on its own merits, don't do it.
- **Understand the motivations of others**: This can be quite tricky, but I believe you should at least think through what might be the motivation of the others around the table. Is that junior partner going to get passed over for promotion if he writes down this deal? Is that other firm fundraising right now? If you don't know, ask. I always aim to be transparent with the entrepreneurs I work with about what my and DFJ's goals and constraints are, independent of my role as a director.

And finally…

- **Understand your own motivation**: What are you doing this for? So you can see your face on the cover of Forbes? So you can have thousands of employees working for you? So you can be a member of the billion dollar Unicorn Club? Perhaps it is to do something you are personally excited about and in a reasonable amount of time, maybe take enough money off the table to live in a nice home, pay for your kid's college and your retirement. I'm not saying one is more correct than the other, I'm just saying that your own goals will dictate whether you should even raise venture at all, how much to raise, and what to spend it on. If you raise $5 million and sell your company for $30 million, it will likely be a life-changing return for you. If you raise $30 million and then sell your company for $30 million, you'll end up like Richard. ■

Heidi Roizen is a venture capitalist, corporate director, Stanford lecturer, recovering entrepreneur and Mom. She co-founded software company T/Maker and served as its CEO for over a dozen years until its acquisition by Deluxe Corporation. After a year as VP of Worldwide Developer Relations at Apple, she became a venture capitalist, and is now the Operating Partner at Silicon Valley-based venture firm DFJ.

Reprinted with permission of the original author.
First appeared in *hn.my/unicorn* (heidiroizen.tumblr.com)

The Days are Long But the Decades are Short

By SAM ALTMAN

I TURNED 30 LAST week and a friend asked me if I'd figured out any life advice in the past decade worth passing on. I'm somewhat hesitant to publish this because I think these lists usually seem hollow, but here is a cleaned up version of my answer:

1. **Never put your family, friends, or significant other low on your priority list.** Prefer a handful of truly close friends to a hundred acquaintances. Don't lose touch with old friends. Occasionally stay up until the sun rises talking to people. Have parties.
2. **Life is not a dress rehearsal — this is probably it.** Make it count. Time is extremely limited and goes by fast. Do what makes you happy and fulfilled — few people get remembered hundreds of years after they die anyway. Don't do stuff that doesn't make you happy (this happens most often when other people want you to do something). Don't spend time trying to maintain relationships with people you don't like, and cut negative people out of your life. Negativity is really bad. Don't let yourself make excuses for not doing the things you want to do.
3. **How to succeed: pick the right thing to do (this is critical and usually ignored), focus, believe in yourself (especially when others tell you it's not going to work), develop personal connections with people that will help you, learn to identify talented people, and work hard.** It's hard to identify what to work on because original thought is hard.
4. **On work: it's difficult to do a great job on work you don't care about.** And it's hard to be totally happy/fulfilled in life if you don't like what you do for your work. Work very hard — a surprising number of people will be offended that you choose to work hard — but not so hard that the rest of your life passes you by. Aim to be the best in the world at whatever you do professionally. Even if you miss, you'll probably end up in a pretty good place. Figure out your own productivity system — don't waste time being unorganized, working at suboptimal times, etc. Don't be afraid to take some career risks, especially early on. Most people pick their career fairly randomly — really think hard about what you like, what fields are going to be successful, and try to talk to people in those fields.
5. **On money: Whether or not money can buy happiness, it can buy freedom, and that's a big deal.** Also, lack of money is very stressful. In almost all ways, having enough money so that you don't stress about paying rent does more to change your wellbeing than having enough money to buy your own jet. Making money is often more fun than spending it, though I personally have never regretted money I've spent on friends, new experiences, saving time, travel, and causes I believe in.
6. **Talk to people more.** Read more long content and fewer tweets. Watch less TV. Spend less time on the Internet.
7. **Don't waste time.** Most people waste most of their time, especially in business.
8. **Don't let yourself get pushed around.** As Paul Graham once said to me, "People can become formidable, but it's hard to predict who." (There is a big difference between confident and arrogant. Aim for the former, obviously.)
9. **Have clear goals for yourself every day, every year, and every decade.**
10. **However, as valuable as planning is, if a great opportunity comes along you should take it.** Don't be afraid to do something slightly reckless. One of the benefits of working hard is that good opportunities will come along, but it's still up to you to jump on them when they do.
11. **Go out of your way to be around smart, interesting, ambitious people.** Work for them and hire them (in fact, one of

the most satisfying parts of work is forging deep relationships with really good people). Try to spend time with people who are either among the best in the world at what they do or extremely promising but totally unknown. It really is true that you become an average of the people you spend the most time with.

12. **Minimize your own cognitive load from distracting things that don't really matter.** It's hard to overstate how important this is, and how bad most people are at it. Get rid of distractions in your life. Develop very strong ways to avoid letting crap you don't like doing pile up and take your mental cycles, especially in your work life.
13. **Keep your personal burn rate low.** This alone will give you a lot of opportunities in life.
14. **Summers are the best.**
15. **Don't worry so much.** Things in life are rarely as risky as they seem. Most people are too risk-averse, and so most advice is biased too much towards conservative paths.
16. **Ask for what you want.**
17. **If you think you're going to regret not doing something, you should probably do it.** Regret is the worst, and most people regret far more things they didn't do than things they did do. When in doubt, kiss the boy/girl.
18. **Exercise.** Eat well. Sleep. Get out into nature with some regularity.
19. **Go out of your way to help people.** Few things in life are as satisfying. Be nice to strangers. Be nice even when it doesn't matter.
20. **Youth is a really great thing.** Don't waste it. In fact, in your 20s, I think it's ok to take a "Give me financial discipline, but not just yet" attitude. All the money in the world will never get back time that passed you by.
21. **Tell your parents you love them more often.** Go home and visit as often as you can.
22. **This too shall pass.**
23. **Learn voraciously.**
24. **Do new things often.** This seems to be really important. Not only does doing new things seem to slow down the perception of time, increase happiness, and keep life interesting, but it seems to prevent people from calcifying in the ways that they think. Aim to do something big, new, and risky every year in your personal and professional life.
25. **Remember how intensely you loved your boyfriend/girlfriend when you were a teenager?** Love him/her that intensely now. Remember how excited and happy you got about stuff as a kid? Get that excited and happy now.
26. **Don't screw people and don't burn bridges.** Pick your battles carefully.
27. **Forgive people.**
28. **Don't chase status.** Status without substance doesn't work for long and is unfulfilling.
29. **Most things are ok in moderation.** Almost nothing is ok in extreme amounts.
30. **Existential angst is part of life.** It is particularly noticeable around major life events or just after major career milestones. It seems to particularly affect smart, ambitious people. I think one of the reasons some people work so hard is so they don't have to spend too much time thinking about this. Nothing is wrong with you for feeling this way; you are not alone.
31. **Be grateful and keep problems in perspective.** Don't complain too much. Don't hate other people's success (but remember that some people will hate your success, and you have to learn to ignore it).
32. **Be a doer, not a talker.**
33. **Given enough time, it is possible to adjust to almost anything, good or bad.** Humans are remarkable at this.
34. **Think for a few seconds before you act.** Think for a few minutes if you're angry.
35. **Don't judge other people too quickly.** You never know their whole story and why they did or didn't do something. Be empathetic.
36. **The days are long but the decades are short.** ■

Reprinted with permission of the original author. First appeared in *hn.my/days* (blog.samaltman.com)

Sam Altman is the President of Y Combinator. He was co-founder and CEO of Loopt, which was funded by Y Combinator in 2005 and acquired by banking company Green Dot in 2012. Mr. Altman also founded Hydrazine Capital. He studied computer science at Stanford University, and while there worked in The Stanford Artificial Intelligence Laboratory.

Lucene: The Good Parts

By ANDREW MONTALENTI

Before MongoDB, before Cassandra, before "NoSQL", there was Lucene.

Did you know that Doug Cutting wrote the first versions of Lucene in 1999? To put things in context, this was around the time Google was more a research project than an actual trusted application. Google's proof-of-concept search engine was still a sprawling set of desktop computers in Stanford's research labs.

I worked on my first Lucene project around 2005. It was a document management system. It didn't have any real issues of scale — it was a web application meant to be run on premise and to provide a view of data that could safely fit in a hard drive or NAS.

But even though the total dataset measured in the hundreds of gigabytes, searching through all the data efficiently was still a challenge. SQL was not then, and is still not now, a very good blob or document storage system. Yet, there seemed to be no alternative to SQL for durability, short of relying directly upon the file system. To boot, the primary use case of the application I was working on was actually document search. People needed to find things. All SQL databases stink at unstructured search, so that's why I started researching Lucene.

Lucene was a Java library you had to learn, and then manually integrate into your app. Thankfully, this wasn't as hopeless as it sounds now.

Among Java projects, Lucene was exceptionally well-documented. Further, Lucene in Action [hn.my/luceneaction] had been published in 2004, and the book went into a lot of depth on how the library worked. I remember purchasing my copy and devouring the book in a weekend. I remember thinking at the time that it was probably one of the best technical books I had read — not just about Lucene, but in general!

A couple of things struck me about Lucene after my first project working with it. First, Lucene approaches problems of data exploration from the vantage point of "information retrieval," not from the vantage point of "database management theory." This meant Lucene was less concerned with things like MVCC, ACID, and 3-NF, and was instead concerned with much more practical concerns, like how to build a fast and humane interface for unstructured data.

Lucene's creator pondered: How do we support queries that normal users will actually type? How do we rapidly search all the data we have, in one fell swoop? How do we order the results when there is more than one likely match? How do we summarize the full result set, even if we only have enough space to display part of the result set?

At the time, Solr and Elasticsearch didn't yet exist. Solr would be released in one year by the team at CNET. With that release would come a very important application of Lucene: faceted search. Elasticsearch would take another 5 years to be released. With its recent releases, it has brought another important application of Lucene to the world: aggregations. Over the last decade, the Solr and Elasticsearch packages have brought Lucene to a much wider community. Solr and Elasticsearch are now being considered alongside data stores like MongoDB and

Cassandra, and people are genuinely confused by the differences.

So, I thought it might be fun to go back to basics. What's so good about Lucene? How does it work under the hood? And why does that give a system like Elasticsearch a leg up on a system like Cassandra in certain applications? Finally, what can we learn from Lucene even if we don't care about full text search?

Jargon terms

So, let's start with a de-jargoning exercise. Here are some terms you see thrown around in the Lucene and Information Retrieval communities which are not nearly as common in the SQL and database communities. Lucene even redefines the term "term" — so, please, pay attention!

- **document**: a record; the unit of search; the thing returned as search results ("not a row")
- **field**: a typed slot in a document for storing and indexing values ("not a column")
- **index**: a collection of documents, typically with the same schema ("not a table")
- **corpus**: the entire set of documents in an index
- **inverted index**: internal data structure that maps terms to documents by ID
- **term**: value extracted from source document, used for building the inverted index
- **vocabulary**: the full set of distinct terms in a corpus
- **uninverted index**: aka "field data": array of all field values per field, in document order
- **doc values**: alternative way of storing the uninverted index on-disk (Lucene-specific)

OK, that gets some jargon out of the way.

Inverting our corpus

Let's start with a simple corpus of two documents, doc1 and doc2. Both contain the field "tag", type "string", with the text "big data". There is also doc3, same structure, but its tag contains the text "small data".

With this small corpus, how can we find things?

Instead of storing:

```
doc1={"tag": "big data"}
doc2={"tag": "big data"}
doc3={"tag": "small data"}
```

We can store the "inverted index". What's that?

```
big=[doc1,doc2]
data=[doc1,doc2,doc3]
small=[doc3]
```

Ah, so it's not an index of documents to terms, it's an index of terms to documents. Clever. If we organize the data this way, we can find documents by value more quickly. When I search for "big", I get back doc1 and doc2. If I search for "small", I get back doc3. If I search for "data", I get back **all documents**. This is basically the core data structure in Lucene and in search in general. Yay for the inverted index!

Not in my vocabulary

In the above documents, I have 3 "terms", and the assumption is that I generated them by doing basic whitespace tokenization. So, my original corpus had the field values `["big data", "small data"]`, but my generated terms are `["big", "small","data"]`.

This already suggests something interesting about terms. If information is repeated in your field values, it will be compressed by pulling out the terms.

By the way, if I were to leave those fields unanalyzed, I'd have two terms: they'd be "big data" and "small data". If I decide **not** to analyze a field, but I decide to store it (in Lucene "stored" field or in Elasticsearch "_source" field), then I am essentially storing the data twice. Once, in the inverted index, and once in the "field storage" (wherever that is) as well.

Terms are interesting when you have data that repeats frequently among your documents. In this small example, the term "data" is repeated in both documents, but only requires one entry in the inverted index. Imagine the same kind of corpus as above, but where you have 1,000 total documents, half tagged with "big data" and half tagged with "small data". In this case you might have:

```
data=[1,2,3,...,1000]
big=[1,3,5,7,9,...,999]
small=[2,4,6,8,...,1000]
```

Here, the inverted index stores one entry for "data", even though data appears in 1,000 documents. It stores one entry for "big", even though it occurs in 500 documents. Likewise for "small".

Big data concordance

You might do a quick back-of-the-envelope calculation at how much more efficient it is to store an inverted index of this data than to store a normal document-to-term index.

Storing the document IDs repeatedly isn't free, but it's certainly cheaper than storing the whole document repeatedly. The vocabulary of a large data set will tend to be much smaller than the record storage of that same data set, and we can take advantage of this at scale.

By the way, this is a pretty ancient technique of mining data. The first complete vocabulary of a complex text was constructed in the year 1262, by 500 **very** patient monks. The document in question was, of course, the Bible, and the vocabulary was called a concordance. How does that proverb go? "There is nothing new under the sun."

Discretely numerical

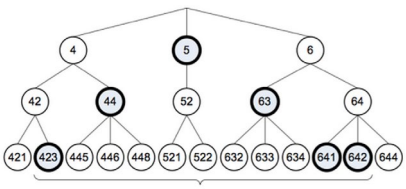
Have a lot of text data you need to make sense of? You clearly have a "search" problem. Have a lot of numeric data you need to make sense of? Well, now, of course, you have an "analytics" problem. Different problem, right?

Well, maybe. The benefits of the inverted index, terms, and vocabularies apply equally well to numeric data. It just requires some lateral thinking to get there.

The reason fields need to have types is because the way we index field values into terms can dramatically affect how we can query those fields. Text is not the only thing that can be broken into terms — numeric and date field values can as well. This is a bit mind-bending, as terms feel like a text-only concept.

Here's a snippet from Lucene in Action on the topic: "If you indexed your field with NumericField, you can efficiently search a particular range for that field using NumericRangeQuery. Under the hood, Lucene translates the requested range into the equivalent set of brackets in the indexed trie structure."

The equivalent set of brackets in the indexed trie structure? Sounds fancy. To start with, what trie structure are we talking about?



An example trie data structure storing numeric data.

Here's an example. Let's suppose I add a new field to my documents called "views". It is a numeric field that contains the number of views each document received on some website. The section above explained how we might find documents that have certain ranges of views, e.g., views between 50 and 100.

If I convert the "views" field into terms, I'll have something that looks like this, perhaps:

```
49=[doc31]
50=[doc40,doc41]
51=[doc53]
...
```

This isn't very helpful. To query for a range of views from 50 to 100, I'd have to construct 50-part query, one for each discrete term:

```
50 OR 51 OR 52 ... OR 100
```

The solution, as mentioned above, is a "trie structure of brackets". Lucene will automatically generate terms that look more like this:

```
49=[doc31]
50=[doc40,doc41]
50x75=[doc40,doc41,doc53,doc78,
doc99,...]
51=[doc53]
...
```

Notice that `50x75` is a special term that encompasses a bracket of 25 discrete values, and thus points to a lot of documents. This allows for smaller queries to cover ranges, and a quicker retrieval of documents over large ranges. The idea is to reduce the discrete numeric data set to a number of lumpier "term ranges". So now, we might be able to cover our 50-100 range with a query like this:

```
50x75 OR 76x99 OR 100
```

The key thing is to select these term ranges automatically — and Lucene has an algorithm for that which ensures that there are enough terms to cover all ranges with good average speed.

Pretty magical, huh? Here's the other clever thing: because numeric values can be converted to term ranges, this same magic works on dates. The dates are converted to numbers, the numbers are then converted into term ranges. Thus, even though you might be searching through 1 million "minutes" of data, you would only be searching through a few hundred "minute ranges" in the inverted index. We could even call these "minute ranges", well, "days"!

The UNIX philosophy introduced the abstraction that "everything is a file", and it certainly required some lateral thinking to make devices like printers and network sockets feel like files. The Lucene philosophy equivalent is, "everything is a term". Numbers, dates, text, identifiers, all can be mapped to behave like text terms, with all the same benefits.

Just uninvert what you've inverted

But, we'd still like to do something with the values within this field. For example, we might like to aggregate up the total views across all of our documents, what in SQL might be a `sum()` aggregate. We might also want to find the document with the most views, that is, to sort our documents by their number of views.

To do this, our inverted index is no help. We might have values ranging from 0 to 100 million in there, with each discrete value (or synthetic range) pointing to the right document IDs. We don't want to find the documents with certain values; we want to instead calculate summaries (aka "analytics") over our corpus or some subset thereof.

A new problem demands another lateral thinking solution. Why don't we uninvert the inverted index? Huh?

In other words, why don't we store, per field, an array of field values, in document order? An index, not of terms to document IDs, but an index of field values that we know (by their order) correspond to specific documents.

```
views=[1,1000,5000,1000000,
       200,...]
```

When we need to do calculations across the whole corpus, we can slurp this array into memory (and, perhaps, keep it there for later). We can then run calculations as fast as computationally possible. If you need to execute a `sum()` on some subset of this array, we can use another trick, Bitsets, for filtering down the array as we go.

The quickest bit

A quick detour. I first heard of Bitsets in one of my favorite programming books, an oldie but goodie passed down to me by my Dad. It's called Programming Pearls.

Its first problem, entitled "Cracking the Oyster," involves solving a specific file sorting problem by representing the lines of the file as an array of bits, where each bit represents one of the possible line values. As described in that chapter, the Bitset is "a dense set over a finite domain when each element occurs at most once and no other data is associated with the element." The author observes that programmers should seek cases where "reducing a program's space requirements also reduces its run time," something he refers to as "mutual improvement." Properly applying a Bitset to a sorting problem is one such example.

Let's return to our uninverted index. Let's say that you want to only sum views from documents that match a specific author. In this case, the full array of views will be compared against a Bitset that might look as follows:

```
views=[1,1000,5000,1000000,
       200,...]
specific_author=[0,1,0,1,0,...]
filtered_views=[0,1000,0,
                1000000,0,...]
```

As you can see, the `views` array was gated through the `specific_author` Bitset, and the result was an array of `filtered_views`. This might even be a sparse array, where most of the values are 0 and the only actual values come from matching documents, but you don't need to worry about that because Lucene uses a compressed Bitset that handles this case nicely.

In any case, this can be done very efficiently in-memory, and the result is a filtered set of field values that matches what we need exactly. Now all we need to do is sum those filtered values.

This makes it clear why it's valuable to have the uninverted index in-memory. Speed. That's why it's often called the field cache.

But in-memory isn't an option for truly big data sets. This leads us to the final chapter.

The solution is obviously… flat files

Storing every single field value in memory is fast but prohibitive. Lucene's "doc values" is basically a hack that takes advantage of Cassandra-style "columnar" data storage.

We store all the document values in a simple format on-disk. Basically, in flat files. Oh, the humanity.

I know what you're thinking. Flat files, how pedestrian! But in this case, we benefit from a few other lateral thoughts. Let's look back at our views array. Rather than storing:

```
views=[1,1000,5000,1000000,
       200,...]
```

We now store the same kind of data in a file that basically just has the values splatted out in column-stride format:

```
1
1000
5000
1000000
200
...
```

We can also be smart and only store binary representation so we can quickly slurp this data into arrays in memory. If the file format on-disk is aligned with the document IDs in our corpus, then we achieve random access to any specific document by seeking into this file. This is the same trick that all columnar, disk-backed key-value stores utilize, for the most part.

When we need to perform a `sum()` on this data, we can simply do a straightforward sequential read of the file. Though it won't put all the data in memory, this scan will signal to the Linux kernel that the disk data is hot, and Linux will start caching.

To quote a kernel developer, "when you read a 100-megabyte file twice, once after the other, the second access will be quicker, because the file blocks come directly from the page cache in memory and do not have to be read from the hard disk again." Flat files, for the win!

Of course, even if the whole file doesn't fit into memory, we can still smartly load large chunks, and know exactly what document range our field values correspond to, thanks to the strict order. This can let us re-use the Bitsets from the earlier section to filter these subsets appropriately.

Lucene: Nice index, OK database

Lucene is not a database — as I mentioned earlier, it's just a Java library. It's coming from the world of information retrieval, which cares about finding and describing data, not the world of database management, which cares about keeping it.

That said, Lucene is an excellent building block for high-performance indices of your data. Solr and Elasticsearch are essentially wrappers on Lucene that use its good parts for information retrieval, and then try to build their own layer atop for persistence. Solr takes advantage of Lucene's built-in "field storage" for this, while Elasticsearch stores JSON blobs inside a Lucene field, called "_source".

Lucene goes even deeper than that, though: using Lucene's API, you can build your own index format (see its Codecs API). Since Lucene's data model is so flexible, when you squint, systems built with Lucene often look like "NoSQL" databases themselves.

With the rise of NoSQL, I've noticed another trend: NIH. No, no, not that NIH ("Not Inverted Here") — I'm talking about **Not Indexed Here**. The rise of MongoDB and Cassandra has also led developers to "roll their own index," mainly out of necessity.

For example, Cassandra encourages you to "determine exactly what queries you need to support," and then store your data in a way to support those queries. Cassandra only really has one index: the partition index. So you have to map all your problems into that one query pattern. Bummer.

Before MongoDB added full text search to its core, it encouraged developers to "use keywords stored in an array in the same document as the text field." Same deal. An indexed keyword array lets you leverage MongoDB's one-trick pony, the BTree index. What's worse, the actual implementation of their built-in full-text search support doesn't introduce any new indexing techniques. It just takes care of generating that keyword's array for you, and then stuffing it in the BTree.

In both of these cases, and in many others, you'd be better off using a Lucene index on your data. Invert your thinking, invert your index. Store your data where you wish, but then build a corpus of Lucene documents with fields corresponding to the data you actually need to find. Anything you put in a field will be indexed and queryable in ad hoc ways. You just need to come to terms with your terms. But, as we've learned, anything can be a term. Convert those into a vocabulary you can actually understand. Then defy comprehension by converting it all into compressed Bitsets. Impress your friends once more by uninverting your inversion. When your sysadmin complains of memory usage, reveal that you've rebuilt the fancy database using none other than flat files. Marvel at how well your OS optimizes for them.

Then, query your Lucene index with pride — a decade-old technology, built on a century of computer science research, and a millennium of monk-like wisdom.

In other words, cutting-edge stuff. ■

Andrew Montalenti is the co-founder & CTO of Parse.ly, a content measurement firm. Parse.ly partners with digital publishers to provide clear audience insights through an intuitive analytics platform. Its engineering team works on time series analytics problems at scale, which is what led them to Lucene.

Reprinted with permission of the original author.
First appeared in *hn.my/lucene* (parsely.com)

151-byte Static Linux Binary in Rust

By KEEGAN MCALLISTER

Part of the sales pitch for Rust [rust-lang.org] is that it's "as bare metal as C."1 Rust can do anything C can do, run anywhere C can run,2 with code that's just as efficient, and at least as safe (but usually much safer).

I'd say this claim is about 95% true, which is pretty good by the standards of marketing claims. A while back I decided to put it to the test by making the smallest, most self-contained Rust program possible. After resolving a few issues along the way, I ended up with a 151-byte, statically linked executable for AMD64 Linux.

Here's the Rust code:

```
#![crate_type="rlib"]
#![allow(unstable)]

#[macro_use] extern crate syscall;

use std::intrinsics;

fn exit(n: usize) -> ! {
    unsafe {
        syscall!(EXIT, n);
        intrinsics::unreachable()
    }
}

fn write(fd: usize, buf: &[u8]) {
    unsafe {
        syscall!(WRITE, fd, buf.as_ptr(), buf.
len());
    }
}

#[no_mangle]
pub fn main() {
    write(1, "Hello!\n".as_bytes());
    exit(0);
}
```

This uses my syscall library [hn.my/syscall], which provides the `syscall!` macro. We wrap the underlying system calls with Rust functions, each exposing a safe interface to the unsafe `syscall!` macro. The main function uses these two safe functions and doesn't need its own unsafe annotation. Even in such a small program, Rust allows us to isolate memory unsafety to a subset of the code.

Because of `crate_type="lib"`, rustc will build this as a static library, from which we extract a single object file `tinyrust.o`:

```
$ rustc tinyrust.rs \
    -O -C no-stack-check -C relocation-
model=static \
    -L syscall.rs/target
$ ar x libtinyrust.rlib tinyrust.o
$ objdump -dr tinyrust.o
0000000000000000 <main>:
   0:   b8 01 00 00 00          mov    $0x1,%eax
   5:   bf 01 00 00 00          mov    $0x1,%edi
   a:   be 00 00 00 00          mov    $0x0,%esi
                        b: R_X86_64_32  .rodata.
str1625
```

```
   f:   ba 07 00 00 00          mov    $0x7,%edx
  14:   0f 05                   syscall
  16:   b8 3c 00 00 00          mov
$0x3c,%eax
  1b:   31 ff                   xor    %edi,%edi
  1d:   0f 05                   syscall
```

We disable stack exhaustion checking, as well as position-independent code, in order to slim down the output. After optimization, the only instructions that survive come from inline assembly blocks in the syscall library.

Note that `main` doesn't end in a `ret` instruction. The `exit` function (which gets inlined) is marked with a "return type" of `!`, meaning "doesn't return". We make good on this by invoking the unreachable intrinsic after syscall!. LLVM [llvm.org]will optimize under the assumption that we can never reach this point, making no guarantees about the program behavior if it is reached. This represents the fact that the kernel is actually going to kill the process before `syscall!(EXIT, n)` can return.

Because we use inline assembly and intrinsics, this code is not going to work on a stable-channel build of Rust 1.0. It will require an alpha or nightly build until such time as inline assembly and `intrinsics::unreachable` are added to the stable language of Rust 1.x.

Note that I didn't even use `#![no_std]`! This program is so tiny that everything it pulls from libstd is a type definition, macro, or fully inlined function. As a result there's nothing of libstd left in the compiler output. In a larger program you may need `#![no_std]`, although its role is greatly reduced following the removal of Rust's runtime.

Linking

This is where things get weird.

Whether we compile from C or Rust,3 the standard linker toolchain is going to include a bunch of junk we don't need. So I cooked up my own linker script [hn.my/linker]:

```
SECTIONS {
    . = 0x400078;

    combined . : AT(0x400078) ALIGN(1) SUB-
ALIGN(1) {
        *(.text*)
        *(.data*)
        *(.rodata*)
        *(.bss*)
    }
}
```

We smash all the sections together, with no alignment padding, then extract that section as a headerless binary blob:

```
$ ld --gc-sections -e main -T script.ld -o pay-
load tinyrust.o
$ objcopy -j combined -O binary payload payload.
bin
```

Finally we stick this on the end of a custom ELF header. The header is written in NASM [nasm.us] syntax but contains no instructions, only data fields. The base address `0x400078` seen above is the end of this header, when the whole file is loaded at `0x400000`. There's no guarantee that ld will put main at the beginning of the file, so we need to separately determine the address of `main` and fill that in as the `e_entry` field in the ELF file header.

```
$ ENTRY=$(nm -f posix payload | grep '^main ' |
awk '{print $3}')
$ nasm -f bin -o tinyrust -D entry=0x$ENTRY
elf.s
$ chmod +x ./tinyrust
$ ./tinyrust
Hello!
```

It works! And the size:

```
$ wc -c < tinyrust
158
```

Seven bytes too big!

The Final Trick

To get down to 151 bytes, I took inspiration from this classic article [hn.my/teensy], which observes that padding fields in the ELF header can be used to store other data. Like, say, a string constant. The Rust code changes to access this constant:

```
use std::{mem, raw};

#[no_mangle]
pub fn main() {
    let message: &'static [u8] = unsafe {
        mem::transmute(raw::Slice {
            data: 0x00400008 as *const u8,
            len: 7,
        })
    };

    write(1, message);
    exit(0);
}
```

A Rust slice like `&[u8]` consists of a pointer to some memory, and a length indicating the number of elements that may be found there. The module `std::raw` exposes this as an ordinary struct that we build, then transmute to the actual slice type. The `transmute` function generates no code; it just tells the type checker to treat our `raw::Slice<u8>` as if it were a `&[u8]`. We return this value out of the `unsafe` block, taking advantage of the "everything is an expression" syntax, and then print the message as before.

Trying out the new version:

```
$ rustc tinyrust.rs \
    -O -C no-stack-check -C relocation-
model=static \
    -L syscall.rs/target
$ ar x libtinyrust.rlib tinyrust.o
$ objdump -dr tinyrust.o
0000000000000000 <main>:
   0:   b8 01 00 00 00          mov    $0x1,%eax
   5:   bf 01 00 00 00          mov    $0x1,%edi
   a:   be 08 00 40 00          mov
$0x400008,%esi
   f:   ba 07 00 00 00          mov    $0x7,%edx
  14:   0f 05                   syscall
  16:   b8 3c 00 00 00          mov
$0x3c,%eax
  1b:   31 ff                   xor    %edi,%edi
  1d:   0f 05                   syscall

...
$ wc -c < tinyrust
151
$ ./tinyrust
Hello!
```

The object code is the same as before, except that the relocation for the string constant has become an absolute address. The binary is smaller by 7 bytes (the size of `"Hello!\n"`) and it still works!

You can find the full code [hn.my/tinyrust] on GitHub. The code in this article works on rustc 1.0.0-dev (44a287e6e 2015-01-08). If I update the code on GitHub, I will also update the version number printed by the included build script.

I'd be curious to hear if anyone can make my program smaller! ■

Keegan is a research engineer at Mozilla, working on Rust and Servo.

Reprinted with permission of the original author.
First appeared in *hn.my/151rust* (mainisusuallyafunction.blogspot.ca)

How I Taught My Dog to Text Me Selfies

By GREG BAUGUES

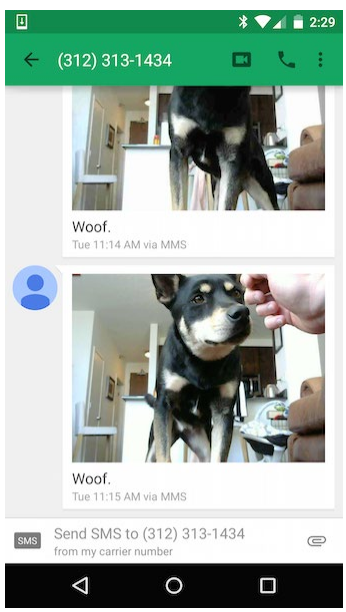
A FEW WEEKS AFTER we got our puppy, we taught her how to turn on a light.

Turns out Kaira will do just about anything if you can clearly communicate your desires and have a treat in your hand. There's an Ikea lamp in our bedroom that's activated by stepping on a floor switch. We started her training by placing her paw on the switch, saying "Light," and giving her a treat. Once she got that, we'd press on her paw and withhold the treat until she heard a click. Eventually, we could say "Light" from across the room and Kaira would run over and do the job: [hn.my/kairalights]

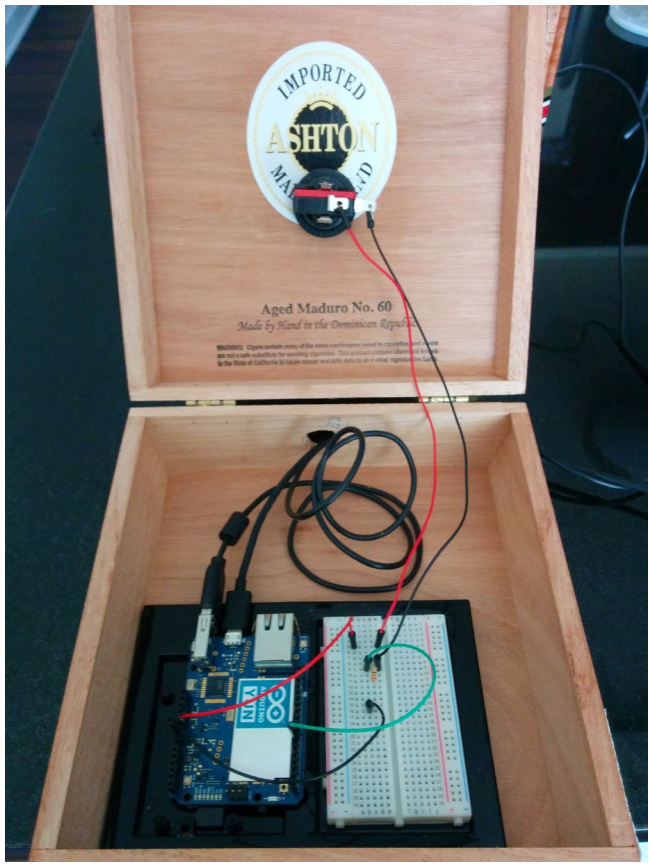
So then I started thinking, "I've got a dog that can press a button. What can I do with that?"

Doggy Selfies

A couple months after Twilio launched MMS, I was reading through Ricky Robinette's post on Training Your Dog with a Tessel [hn.my/tessel] and started to wonder if we could teach Kaira to send selfies. I'm pleased to say that, thanks to the Arduino Yun and a big red button, the answer is a resounding "Yes!" as you can see in this short video: [hn.my/kairaselfie]



My dog texts me a selfie



What you're seeing in the video is a cigar box housing a massive arcade button and an Arduino Yun. The second box serves as a stand for a webcam that's plugged into the Yun. (My local cigar shop sells empties for $2 which make for sturdy and stylish hardware enclosures).

The Wifi-enabled Arduino Yun has two microprocessors: one does all the pin interaction you typically associate with an Arduino. The second runs a stripped-down version of Linux called OpenWRT which can run programs in your favorite scripting language. (Python comes pre-installed, but you could put Ruby or Node on there if you so please.) This project has one program running on each processor. Together, they are less than 60 lines of code.

The Arduino sketch simply:

- Waits for a button press
- Runs a shell command to take a picture
- Runs a Python script to upload the picture to Dropbox and send the MMS

```
#include <Bridge.h>
#include <Process.h>

const int BUTTON = 7;

void setup() {
  pinMode(BUTTON, INPUT);
  Bridge.begin();
}

void loop() {
  if (digitalRead(BUTTON) == HIGH) {
    takePicture();
    uploadAndSend();
  }
}

void takePicture() {
  Process p;
  p.begin("fswebcam");
  p.addParameter("/mnt/sda1/pic.jpg");
  p.addParameter("-r 640x480");
  p.run();
}
```

```
void uploadAndSend() {
  Process p;
  p.begin("python");
  p.addParameter("/mnt/sda1/arduino/upload-and-
send.py");
  p.run();
}
```

The Python script uses the Dropbox SDK and Twilio helper library to:

- Upload the picture to Dropbox
- Get a publicly accessible url for the picture
- Use that url to send an MMS via Twilio

```
import datetime
import dropbox
from twilio.rest import TwilioRestClient

dropbox_access_token = "YOURDROPBOXTOKEN"
twilio_phone_number = "YOURTWILIOPHONENUMBER"
twilio_account_sid = "YOURTWILIOACCOUNTSID"
twilio_auth_token = "YOURTWILIOAUTHTOKEN"
cellphone = 'YOURCELLPHONE'

timestamp = datetime.datetime.now().
strftime("%h-%m-%S")
filename = "kaira-" + timestamp + ".jpg"

f = open("/mnt/sda1/pic.jpg")
dropbox_client = dropbox.client.
DropboxClient(dropbox_access_token)
response = dropbox_client.put_file(filename, f)
url = dropbox_client.media(response['path'])
['url']

twilio_client = TwilioRestClient(twilio_account_
sid, twilio_auth_token)
twilio_client.messages.create(
  to = cellphone,
  from_ = twilio_phone_number,
  body = "Woof.",
  media_url = url)
```

If you'd like some more color on how we got here, we've documented the entire process from Arduino Yun unboxing to sending MMS in these three tutorials:

- Getting Started with the Arduino Yun [hn.my/yunstart]
- Take a picture with a webcam and upload it to Dropbox from the Yun [hn.my/yuncam]
- Send SMS and MMS from your Arduino Yun [hn.my/yunsms]

Onward!

What's most exciting to me about this project, aside from the sheer novelty of my dog sending selfies, is how simple each component is. The button press is literally the second example from Massimo Banzi's Getting Started with Arduino. [hn.my/banzi]

The Python script is practically cut-and-paste from the Dropbox and Twilio getting started guides.

Hardware hacking can be intimidating if you've never done it before, but remember that the most impressive hacks are often just simple building blocks stacked on top of one another. Wifi-enabled devices like the Arduino Yun have drastically lowered the barrier to entry for web developers to dip their toes into the Internet of Things.

So let's say you had a box that could interact with both the physical world and any web-based API using the programming language you already know. What could you do with that? ■

Greg Baugues serves as a developer evangelist at Twilio, a company that empowers developers to integrate text messaging, phone calling, and video conferencing into their apps with just a few lines of code. You can find him on Twitter at *@greggyb*

Reprinted with permission of the original author.
First appeared in *hn.my/selfies* (twilio.com)

Main Is Usually a Function. So Then When Is It Not?

By JAMES ROWE

It began when my coworker, despite already knowing how to program, was forced to take the intro level Computer Science course at my university. We joked with him about how he needs to make a program that works, but the grading TAs wouldn't be able to figure out how it works. So that is the requirement, to make a functioning program that completes an assignment while obfuscating it such that the graders think that it shouldn't work. With this in mind, I started to think through the arsenal of tricks in C that I've seen used before and one thing in particular stood out. The idea for this trick I will explain how to accomplish came from a blog with the name main is usually a function [mainisusuallyafunction.blogspot.ca] which got me thinking about when would main not be a function? Let's find out then!

My problem-solving process is typically the same thing I imagine most programmers go through.

- **Step 1**: Google search about the problem.
- **Step 2**: Click every link on the first page that seems relevant. If not solved, try a different query and repeat.

Thankfully, the answer to this question came on the very first search on this Stack Overflow answer [hn.my/somain]. Apparently in 1984, a strange program won the IOCCC where main was declared as a `short main[] = {...}` and somehow this did stuff and printed to the screen! Too bad it was written for a whole different architecture and compiler, so there is really no easy way for me to find out what it did. But judging from the fact that it is just a bunch of numbers, I can surmise that the numbers there are just the compiled binary of some short function, and the linker, when looking for the main function, just throws this in its place.

With our hypothesis in place (that the code for the program is just the compiled assembly of main function represented as an array), let's see if we can replicate this by making a small program.

```
char main[] = "Hello world!";
$ gcc -Wall main_char.c -o first
main_char.c:1:6: warning: 'main' is usually a
function [-Wmain]
 char main[] = "Hello world!";
      ^
$ ./first
Segmentation fault
```

All right! It worked! Kind of…. So our next goal is to actually print something to the screen. Thinking back to my limited ASM experience, I recalled that there are different sections of the compiler which determine where different things go. The two sections that are most relevant to us are the `.text` section and the `.data` section. `.text` contains all the executable code, and it is read-only, whereas `.data` contains readable and writable code, but isn't executable. In our case, we can only fill in code for the main function, so anything that gets placed in the data section is a no-go. We need to find a way to get the string `"Hello world!"` inside the main function and reference it.

I began by looking into how to print something with as little code as possible. Since I knew the target system is going to be 64bit Linux, I found that I could call the system, write call, and it would write out to the screen. Looking back at this now that I'm writing the code, I don't think that I needed to use Assembly for this, but at the same time, I'm really glad I got to learn what I did. Getting started writing inline GCC ASM was the hardest part, but once I got the hang of it, it started to become easier.

Getting started wasn't easy though. It turns out that most of the ASM knowledge I could find through Google is all of the following: really old, Intel syntax, and for 32 bit systems. Remember in our scenario, we need the file to compile with a GCC on a 64 bit system, without any special modifications to the compiler flags. That means there are no special compile flags, nor can we include any custom linking steps. Plus, we want to use GCC inline AT&T syntax. Most of my time was spent trying to find information about modern assembly for 64 bit systems! Maybe my Google-foo is lacking.

This part was almost all trial and error. My goal was just to use the write syscall to print "Hello world!" to the screen using GCC inline ASM, so why was it so hard? For the people that want to learn how to do this, I recommend the following sites: Linux syscall list [hn.my/syscalltable], Intro to Inline Asm [hn.my/inlineasm], and Differences between Intel and AT&T Syntax [hn.my/linasm].

Eventually my ASM code started to form, and I had some code that seemed to work! Remember, my goal is to produce a main that is an array of the ASM that prints Hello World.

```
void main() {
    __asm__ (
        // print Hello World
        "movl $1, %eax;\n"  /* 1 is the syscall
number for write on 64bit */
        "movl $1, %ebx;\n"  /* 1 is stdout and
is the first argument */
        "movl $message, %esi;\n" /* load the
address of string into the second argument*/
        "movl $13, %edx;\n"  /* third argument
is the length of the string to print*/
        "syscall;\n"
        // call exit (so it doesn't try to run
        // the string Hello World)
        // maybe I could have just used ret
        // instead?
        "movl $60,%eax;\n"
        "xorl %ebx,%ebx; \n"
        "syscall;\n"
        // Store the Hello World inside the main
        // function
        "message: .ascii \"Hello World!\\n\";"
    );
}
$ gcc -Wall asm_main.c -o second
asm_main.c:1:6: warning: return type of 'main'
is not 'int' [-Wmain]
 void main() {
      ^
$ ./second
Hello World!
```

Hurray! It prints! Let's take a look at the compiled code in hex now, and it should match up 1 to 1 with the ASM code we wrote. I went ahead and broke down what's going on in the comments to the side.

```
(gdb) disass main
Dump of assembler code for function main:
   0x00000000004004ed <+0>:     push   %rbp
; Compiler inserted
   0x00000000004004ee <+1>:     mov    %rsp,%rbp
   0x00000000004004f1 <+4>:     mov    $0x1,%eax
; It's our code!
   0x00000000004004f6 <+9>:     mov    $0x1,%ebx
   0x00000000004004fb <+14>:    mov
$0x400510,%esi
   0x0000000000400500 <+19>:    mov    $0xd,%edx
   0x0000000000400505 <+24>:    syscall
   0x0000000000400507 <+26>:    mov
$0x3c,%eax
   0x000000000040050c <+31>:    xor    %ebx,%ebx
   0x000000000040050e <+33>:    syscall
   0x0000000000400510 <+35>:    rex.W
; String hello world
   0x0000000000400511 <+36>:    gs
; it's garbled since
   0x0000000000400512 <+37>:    insb
(%dx),%es:(%rdi) ; it's not real asm
   0x0000000000400513 <+38>:    insb
(%dx),%es:(%rdi) ; so it couldn't be
   0x0000000000400514 <+39>:    outsl
%ds:(%rsi),(%dx) ; disassembled
   0x0000000000400515 <+40>:    and
```

```
%dl,0x6f(%rdi)
   0x0000000000400518 <+43>:    jb     0x400586
   0x000000000040051a <+45>:    and
%ecx,%fs:(%rdx)
   0x000000000040051d <+48>:    pop    %rbp
; Compiler inserted
   0x000000000040051e <+49>:    retq
End of assembler dump.
```

That looks like a functioning main to me! Now let's go and grab the hex contents of it, and dump it in as a string and see if that works. We can get the hex from main by using gdb again. I'm willing to guess that there must be a better way. The way I did it was to load gdb and print the hex at main, like so. Last time we disassembled main, we saw that it was 49 bytes long, so we can use the `dump` command to save the hex to a file.

```
# example of how to print the hex
(gdb) x/49xb main
0x4004ed <main>:    0x55    0x48    0x89    0xe5
0xb8    0x01    0x00    0x00
0x4004f5 <main+8>:  0x00    0xbb    0x01    0x00
0x00    0x00    0xbe    0x10
0x4004fd <main+16>: 0x05    0x40    0x00    0xba
0x0d    0x00    0x00    0x00
0x400505 <main+24>: 0x0f    0x05    0xb8    0x3c
0x00    0x00    0x00    0x31
0x40050d <main+32>: 0xdb    0x0f    0x05    0x48
0x65    0x6c    0x6c    0x6f
0x400515 <main+40>: 0x20    0x57    0x6f    0x72
0x6c    0x64    0x21    0x0a
0x40051d <main+48>: 0x5d
# example of how to save it to a file
(gdb) dump memory hex.out main main+49
```

Now that we have the hex dump, we can convert them all to integers the easiest way that I know how: using Python. In Python 2.6 and 2.7, you can just use the following to convert it to a convenient array of ints for us to use.

```
>>> import array
>>> hex_string = "554889E5B801000000BB-
01000000BE10054000BA0D0000000F05B83C00000031DB-
0F0548656C6C6F20576F726C64210A5D".decode("hex")
>>> array.array('B', hex_string)
array('B', [85, 72, 137, 229, 184, 1, 0, 0, 0,
187, 1, 0, 0, 0, 190, 16, 5, 64, 0, 186, 13, 0,
0, 0, 15, 5, 184, 60, 0, 0, 0, 49, 219, 15, 5,
72, 101, 108, 108, 111, 32, 87, 111, 114, 108,
100, 33, 10, 93])
```

I figure if my bash foo and Unix knowledge was greater, I could find an easier way to do this, but Googling things like "hex dump of compiled function" returns several questions about how to print hex in various languages. Regardless, we now have a comma-separated array of our function, so let's put that in a new file and see if it works! I went ahead and commented what each of the different values mean.

```
char main[] = {
    85,                 // push   %rbp
    72, 137, 229,       // mov    %rsp,%rbp
    184, 1, 0, 0, 0,    // mov    $0x1,%eax
    187, 1, 0, 0, 0,    // mov    $0x1,%ebx
    190, 16, 5, 64, 0,  // mov    $0x400510,%esi
    186, 13, 0, 0, 0,   // mov    $0xd,%edx
    15, 5,              // syscall
    184, 60, 0, 0, 0,   // mov    $0x3c,%eax
    49, 219,            // xor    %ebx,%ebx
    15, 5,              // syscall
    // Hello world!\n
    72, 101, 108, 108, 111, 32, 87, 111, 114,
108, 100,
    33, 10,             // pop    %rbp
    93                  // retq
};
$ gcc -Wall compiled_array_main.c -o third
compiled_array_main.c:1:6: warning: 'main' is
usually a function [-Wmain]
 char main[] = {
      ^
$ ./third
Segmentation fault
```

Segfault! What am I doing wrong? Time to fire up gdb again and try to see what the error is. Since main is no longer a function, we can't simply use `break main` to set a break point there. Instead, we can use `break _start` to get a breakpoint at the method that calls the libc runtime startup (which in turn calls main), and we can see what address we pass to `__libc_start_main`.

```
$ gdb ./third
(gdb) break _start
(gdb) run
(gdb) layout asm
```

```
   ──────────────────────────────────
B+>|0x400400 <_start>    xor    %ebp,%ebp
   |0x400402 <_start+2>  mov    %rdx,%r9
   |0x400405 <_start+5>  pop    %rsi
   |0x400406 <_start+6>  mov    %rsp,%rdx
   |0x400409 <_start+9>  and
$0xfffffffffffffff0,%rsp
   |0x40040d <_start+13> push   %rax
   |0x40040e <_start+14> push   %rsp
   |0x40040f <_start+15> mov    $0x400560,%r8
   |0x400416 <_start+22> mov    $0x4004f0,%rcx
   |0x40041d <_start+29> mov    $0x601060,%rdi
   |0x400424 <_start+36> callq  0x4003e0 <__
libc_start_main@plt>   |0x400424 <_start+36>
callq  0x4003e0 <__libc_start_main@plt> |
```

From testing, I found that the value pushed on `%rdi` is the location of main, but something seems off this time. Hang on, it put main in the `.data` section! Earlier I mentioned how `.text` is where read-only executable code goes and `.data` is where non-executable read/write values go! The code is trying to run memory that is marked as non-executable, which is the cause of the segfault. How am I supposed to convince the compiler that my "main" belongs in `.text`?! Well, my searches turned up empty, and I was convinced that was the end of the road. Time to call it a night and deem my adventure a failure.

But I couldn't sleep that night without finding a solution. I continued to search and search some more until I found a very obvious and simple solution on a Stack Overflow post. All I have to do is declare the main function as const. Changing it to `const char main[] = {` was all I needed to do to get it in the right section. Let's try compiling again.

```
$ gcc -Wall const_array_main.c -o fourth
const_array_main.c:1:12: warning: 'main' is usu-
ally a function [-Wmain]
 const char main[] = {
            ^
$ ./fourth
SL)1HH
```

Ack! What is it doing now? Time to gdb again and see what's happening.

```
gdb ./fourth
(gdb) break _start
(gdb) run
(gdb) layout asm
```

So looking at the code, we can see the address for main is in the ASM for _start in the instruction that looks like this on my machine `mov $0x4005a0,%rdi`. We can use this to set a break point on main by doing `break *0x4005a0` and then continue execution with `c`.

```
(gdb) break *0x4005a0
(gdb) c
(gdb) x/49i $pc     # $pc is the current
                    # executing instruction
...
   0x4005a4 <main+4>:   mov    $0x1,%eax
   0x4005a9 <main+9>:   mov    $0x1,%ebx
   0x4005ae <main+14>:  mov    $0x400510,%esi
   0x4005b3 <main+19>:  mov    $0xd,%edx
   0x4005b8 <main+24>:  syscall
...
```

I snipped some of the assembly that wasn't important. If you didn't notice what went wrong, the address pushed to print at (0x400510) is not the address we stored the string "Hello world!\n" at (0x4005c3)! It's actually still pointing to the computed location in the original compiled executable and isn't using relative addressing to print it. That means we need to modify the assembly code in order to load the address of the string relative to the current address. As it stands, it's fairly difficult to accomplish in 32 bit code, but thankfully we are using 64bit ASM, so we can use the `lea` instruction to make it easier.

```
void main() {
    __asm__ (
        // print Hello World
        "movl $1, %eax;\n"  /* 1 is the syscall
number for write */
        "movl $1, %ebx;\n"  /* 1 is stdout and
is the first argument */
        // "movl $message, %esi;\n" /* load the
        // address of string into the second
        // argument*/ instead use this to load
        // the address of the string as 16 bytes
        // from the current instruction
        "leal 16(%eip), %esi;\n"
        "movl $13, %edx;\n"  /* third argument
is the length of the string to print*/
        "syscall;\n"
        // call exit (so it doesn't try to run
        // the string Hello World maybe I could
        // have just used ret instead
        "movl $60,%eax;\n"
        "xorl %ebx,%ebx; \n"
        "syscall;\n"
        // Store the Hello World inside the main
        // function
        "message: .ascii \"Hello World!\\n\";"
    );
}
```

The changed code is commented so you can see it. Compiling the code and checking to see if it works:

```
$ gcc -Wall relative_str_asm.c -o fifth
relative_str_asm.c:1:6: warning: return type of
'main' is not 'int' [-Wmain]
 void main() {
      ^
$ ./fifth
Hello World!
```

And now we can use the same techniques discussed earlier to extract the hex values as an integer array. But this time, I want to make it a little bit more disguised and tricky by using the full 4 bytes that ints give me instead. We can do that by printing the information out in gdb as an int instead of dumping the hex to a file and then copy/pasting it into the program.

```
gdb ./fifth
(gdb) x/13dw main
0x4004ed <main>:    -443987883  440 113408
-1922629632
0x4004fd <main+16>: 4149    899584  84869120
15544
0x40050d <main+32>: 266023168   1818576901
1461743468  1684828783
0x40051d <main+48>: -1017312735
```

I chose the number 13 since main was 49 bytes long, and 49 / 4 rounds up to 13 just to be safe. Since we exit from the function early, it shouldn't make a difference. Now all that's left is to copy and paste this back into our `compiled_array_main.c` and run it.

```
const int main[] = {
    -443987883, 440, 113408, -1922629632,
    4149, 899584, 84869120, 15544,
    266023168, 1818576901, 1461743468,
1684828783,
    -1017312735
};
$ gcc -Wall final_array.c -o sixth
final_array.c:1:11: warning: 'main' is usually a
function [-Wmain]
 const int main[] = {
           ^
$ ./sixth
Hello World!
```

And all this time we've been ignoring the warning message about main not being a function!

I'm guessing all that will happen when my coworker turns in an assignment looking like this is they will take off points for bad coding style and say nothing else about it. ■

James Rowe is a senior in Computer Science that loves to explore his interests and doing whatever he thinks is cool. Hobbies include gaming, game programming, web programming, and anything in between. He is currently looking for a job to start after he graduates.

Reprinted with permission of the original author.
First appeared in *hn.my/main* (jroweboy.github.io)

Four Days of Go

By EVAN MILLER

Part of my work involves the mild reverse-engineering of binary file formats. I say "mild" because usually other people do all of the actual work; I just have to figure out what an extra flag field or two means, and I then take as much credit as possible for the discovery on my blog.

To see what's in the guts of a binary file, I use a hex editor, though even my favorite one [synalysis.net] is a bit of a chore to use. When I'm trying to figure out a file format, I want to mark it up with my hypotheses about what various bytes may mean, but currently there aren't any hex editors that will let me do that. My workflow at present is to print out the hex representation of a binary file onto physical sheets of paper, and then mark them up with a ball-point pen that I received last year at a conference about technology.

To save a few trees, and to ensure that my Conference Pen Collection remains in pristine condition for a future eBay auction, I decided to write my own hex editor suited for reverse-engineering tasks. I've had a hex-editor name picked out for a while now (*Hecate: The Hex Editor From Hell*), as well as a color palette and appropriate thematic iconography (think Dante's *Inferno* meets Scorsese's *Taxi Driver*).

I also had some visual ideas for the program worked out, but before I could get serious about tinkering, I realized I needed to choose a development platform. I use three platforms on a regular basis (OS X, a terminal, and the World Wide Web), so I decided to organize a three-way imaginary cage fight between them, i.e., construct a list of pros and cons for each.

I know OS X pretty well at this point, and I thought about writing the program in Swift. However, I wanted to make Hecate cross-platform and open-source, so that other people could contribute to the project without me having to pay them. A browser version could make sense, but I'd rather not spend my time running a Hex Editing Web Service, nor do I want my users babysitting a local Node.js instance or whatever on their computer.

I spent a few minutes considering a cross-platform C++ toolkit such as Qt. Then the police arrived and told me in a calming manner to put down the hunting rifle, so that left me with the last (and original) computing platform: the terminal.

Apparently terminal applications have been experiencing something of a retro chic Renaissance, driven by the California New Wave of systems programming languages. I still enjoy programming in old-school, Jersey-style, scorched-earth C, and gave some thought to writing Hecate in it, but I was assured by several GitHub pages that the `ncurses` library is a horrible macro-infested mess, and I decided to explore other options.

There's actually a new terminal library written in C called termbox; that was my first choice, but then I saw the author mention that the Go version of the library had more features. More features, of course, are always a good thing to have, especially in a library you've never used before, so I thought, what the hay, let's learn a new programming language.

Hello, Go: First impressions

When I program I usually think in C, that is, as I type I try to think about the C code that's actually being executed when the program runs. I tend to prefer languages where I have a reasonable chance of understanding what will be executed; but I also appreciate being able to throw caution to the

garbage collector and bang out code in a hurry when the occasion calls for it.

Go code will look at least a little familiar to C programmers. It carries over C's primitive types, as well as its semantics with regard to values and pointers. From the start I had a decent mental model of how things are passed to functions in Go, and under what circumstances the caller can expect data to change. Like C, Go shuns classes in favor of plain structs, but then lets you make code more object-oriented with interfaces, which I'll discuss in a minute.

First let's talk about the basic syntax. Go is statically typed with type inference, which saves some typing, and it splits out the declaration/initialization and assignment operators into `:=` and `=`, like this:

```
my_counter := 1
// an exciting new variable

my_counter = 2
// update the variable

my_counter := 3
// this produces an error
```

Although when I started I wasn't accustomed to the `=` versus `:=` distinction, I began to like it as a way to catch editing errors. Functions can have multiple return values, but the rules for multiple assignment feel a little odd; the left-hand side of a multiple assignment is allowed to have a mix of declared and undeclared variables, but you need to use `:=` when there is at least one undeclared variable on the left. That is to say:

```
my_counter := 1
// an exciting new variable

my_counter, _ = update_
counter(my_counter)
// OK

my_counter, _ := update_
counter(my_counter)
// not OK

// The following line is OK.
// Even though my_counter
// already exists, error is a
// new variable, so := is
// appropriate
my_counter, error :=
update_counter(my_counter)
```

To me it seems a little arbitrary that the `:=` should somehow dominate the `=`, and it also introduces room for the very bugs that `:=` was supposed to prevent (e.g., I might think I am declaring `my_counter` in two places).

I think a more logical syntax would be to have the number of colons equal to the number of new left-hand variables (`::=` for a `double` declaration, `:::=` for a triple declaration, etc.), but I guess the language designers couldn't find my phone number at the critical moment during Go's research and development phase.

Go has eliminated some traditional keywords in favor of overloading `if` and `for`. Go's `for` can be used in place of C's `for`, `while`, and `while(1)`, and there is a two-statement version of `if` that I just found out about yesterday. I suppose these consolidations technically make the language simpler, but it also makes code slightly harder to talk about. When looking at someone's C code, I can say "use a while loop here instead of a for loop," but with Go I would have to say "use a zero-statement for loop instead of a three-statement for loop". It is possible, however, that the Go team has developed a set of secret signifiers to distinguish these constructions in everyday conversation and not told me about them. (Did you know that the `$_` variable in Perl is called "it"? I read that in a book. If I remember correctly the name comes from a Stephen King novel.)

Go has also eliminated the ternary operator, and, for reasons that appear to be political, does not have integer Min and integer Max functions. From what I can gather on the mailing list threads, the language designers are against polymorphism, as well as adding letters to function names, so unlike the standard C library which operates on `float`, `double`, and `long double`, as well as `int` and long where appropriate (e.g., absolute value), the Go standard math library operates only on `float64`. Since there's no implicit casting to `float`s, this is rather annoying if you're using integers, such as when you are counting things. It also makes Go somewhat less useful for heavy number-crunching where you might want single- or extended-precision versions of `float`ing-point functions.

(Incidentally, the only language I know that gets polymorphism right for dealing with multiple kinds of mathematical objects is Julia — though last time I checked it was still lacking `long double` / `float80` support.)

By the way, if anyone who works on the Go math library is reading this, there are a few important functions missing. [hn.my/gomath]

The rest of the standard library looks good to me so far — I like the design of the string-formatting library, and the Unicode support is excellent. `rune` is an odd way to name your character type, but I suppose they wanted to avoid confusion with C's 8-bit `char`. (In English usage, *rune* refers specifically to a character from a medieval Germanic alphabet, or a glyph believed to have magical powers. While some people might object to the character type having mystical connotations in Go, I fully support all references to medieval texts and/or the occult in programming languages.)

Go is "OO-ish" with its use of interfaces — interfaces are basically duck typing for your structs (as well as other types, because, well, just because). I had some trouble at first understanding how to get going with interfaces and pointers. You can write methods that act on WhateverYouWant — and an interface is just an assertion that WhateverYouWant has methods for X, Y, and Z. It wasn't really clear to me whether methods should be acting on values or pointers. Go sort of leaves you to your own devices here.

At first I wrote my methods on the values, which seemed like the normal, American thing to do. The problem of course is that when passed to methods, values are copies of the original data, so you can't do any OO-style mutations on the data. So instead methods should operate on the pointers, right?

This is where things get a little bit tricky if you're accustomed to subclassing. If you operate on pointers, then your interface applies to the pointer, not to the struct (value). So if in Java you had a `Car` with `RaceCar` and `GetawayCar` as subclasses, in Go you'll have an interface `Car` — which is implemented not by `RaceCar` and `GetawayCar`, but instead by their pointers `RaceCar*` and `GetawayCar*`.

This creates some friction when you're trying to manage your car collection. For example, if you want an array with values of type `Car`, you need an array of pointers, which means you have to need separate storage for the actual `RaceCar` and `GetawayCar` values, either on the stack with a temporary variable or on the heap with calls to `new`. The design of interfaces is consistent, and I generally like it, but it had me scratching my head for a while as I got up to speed with all the pointers to my expensive and dangerous automobiles.

Go is garbage-collected. I personally think Swift/Objective-C-style Automatic Reference Counting would have been a better choice for a statically typed systems language, since it gives you the brevity benefits without the GC pauses. I'm sure this has been argued to death elsewhere, so I'll save my GC rant for a very boring dinner party.

One of Go's major selling points is its concurrency support. I have not yet played with its concurrency features, cutely called goroutines. My impression from the description is that while goroutines are an advancement over vanilla C and C++, Go lacks a good story for handling programmer errors in a concurrent environment. Normal errors are bubbled up as values, but if there's a programmer error (e.g., index out of range), the program panics and shuts down.

For single-threaded programs, this is a reasonable strategy, but it doesn't play very well with Go's concurrency model. If a goroutine panics, either you take down the whole program, or you recover — but then your shared memory may be left in an inconsistent state. That is, Go assumes programmers will not make any mistakes in the recovery process — which is not a very good assumption, since it was programmer error that brought about the panic in the first place. As far as I know, the only language that really gets this right is Erlang, which is designed around shared-nothing processes, and thus programmer errors are properly contained inside the processes where they occur.

(It's also worth mentioning that you can get Go-style M:N concurrency model in C by using Apple's libdispatch [libdispatch.macosforge.org]. In conjunction with block syntax, it's a fairly nice solution, though like Go, it's not robust to programmer error.)

I had previously read about Go's refusal to compile programs with unused import statements, but I didn't really believe it until, well, I couldn't compile a Go program that contained an unused import statement. (The same goes for unused variables.) The Go FAQ gets a bit pedantic on this point — explaining to you why it's for your own good — but in practice, it makes the language less fun to tinker with. I prefer to try things out and get them working, then go back later and clean things up. Go basically forces you to have clean code all along, which is a bit like forcing a scientist to wipe down the workbench and rinse all the beakers between every experiment, or forcing a writer to run the spell checker after every cigarette. It sounds like good practice, but it comes with a cost, and it's a decision that's

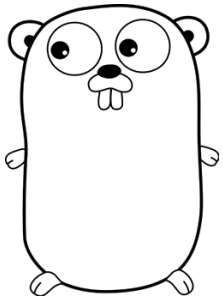
probably best left to the person it immediately affects, rather than to the tool designers.

As an aside, I personally would like to see a version of Go called "Sloppy Go" that will only compile programs that contain at least one unused import and several unused variables, and maybe an unmatched parenthesis, just to ensure that the programmer still knows how to have fun.

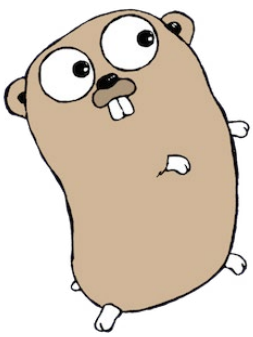
I was trying to think of why the Go designers thought it was such a good idea to refuse to compile programs with unused variables. I have a theory, and will take a detour here into what I believe to be the psychological foundations of the Go programming language. I call it the Autistic Gopher Hypothesis.

The Autistic Gopher Hypothesis

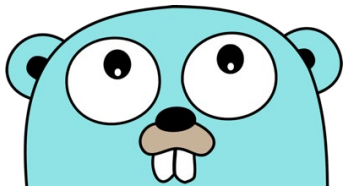
I didn't mention the *very* first impression I had of Go. On the Go homepage, there is a gopher — the language mascot — facing you. But he's looking to the left.



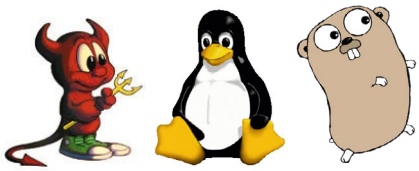
Other times he's looking to the right.



Even when he's looking in your direction, it's like he's looking slightly upwards, perhaps at your toupée.



There was always something a little unsettling to me about the Go gopher. He's always moving around and never quite makes eye contact with the viewer. Compared to the Go gopher, a devil looks downright approachable. Even a penguin looks warm:



The Go gopher doesn't look dangerous per se, but doesn't he seem a little… odd? He faces you head-on as if he wants your attention and approval, but he's not engaging you, and certainly not listening to you. If I had to guess, I'd say the Go gopher suffers from a mild form of autism.

I get the same feeling about the Go language. It feels like it is designed by an obsessive personality — obsessed with build times in particular, but also having an obsession with detail, someone who rarely makes mistakes when writing code, who generally will not run code until it appears to be complete and correct.

Normally I'd appreciate these qualities in a compiler writer, but I feel that the designer went too far, to the point of being antisocial, i.e., attempting to impose arbitrary rules on the language users. I imagine that this person is tired of dealing with warning-riddled code produced by colleagues — code full of unused variables and imports, slow-building code that takes up the designer's precious time — and has decided to exert control over the type of code written by colleagues not by the normal organizational and political processes (e.g., lobbying for `-Wall -Werror` on the build server), but by producing a compiler that refuses any input that doesn't meet the designer's own exacting standards for computer code. The designer realizes that giving any ground, e.g., having compiler warnings of any kind, creates a potential political battle within the designer's organization. Thus the designer has circumvented the normal give-and-take over the build server configuration simply by eliminating flags from the compiler.

In other words, Go represents a kind of Machiavellian power play, orchestrated by slow-and-careful programmers who are tired of suffering for the sins of fast-and-loose programmers. The Go documentation refers quite often to intolerable 45-minute build times suffered by the original designers, and I can't help but imagine them sitting around and seething about all those unused imports from those "other" programmers, that is, the "bad" programmers. Their solution was not to engage and educate those programmers to change their habits, but rather design a new language that the bad programmers would be compelled to use — and tie down the language sufficiently so that "bad" practices, such as a program containing unused variables, were impossible.

Reading Go's mailing list and documentation, I get a similar sense of refusal-to-engage — the authors are communicative, to be sure, but in a didactic way. They seem tired of hearing people's ideas, as if they've already thought of everything, and the relative success of Go at Google and elsewhere has only led them to turn the volume knob down. Which is a shame, because they'll probably miss out on some good ideas (including my highly compelling, backwards-incompatible, `double`-triple-colon-assignment proposal mentioned above).

Under this theory, more of the language choices start to make sense. There is no ternary operator because the language designers were tired of dealing with other people's use of ternary operators. There is One True Way To Format Code — embodied in gofmt — because the designers were tired of how other people formatted their code. Rather than debate or engage, it was easier to make a new language and shove the new rules onto everyone by coupling it with Very Fast Build Times, a kind of veto-proof Defense Spending Bill in the Congress of computer programming. In this telling, the story of Go is really a tale of revenge, not just against slow builds, but against all kinds of sloppy programming.

Which in my opinion is too bad, because I myself am a sloppy programmer. I love writing sloppy code. Not because I like having sloppy code, or maintaining sloppy code — but because I like to tinker and play with code. I like trying a bunch of different library calls to see exactly what they do. I like trying a bunch of interface ideas and seeing which works best. The faster I can get results from my code, the faster I can understand the problem at hand. For me, writing code is as much about acquiring knowledge as it is about producing something of lasting value. So in the process of writing code, I'll leave behind a wasteland of fallow variables and futile imports, but I don't really care, because there's a good chance I'll throw away the whole file anyway. Frankly, my unused variables are none of anyone's damn business but my own.

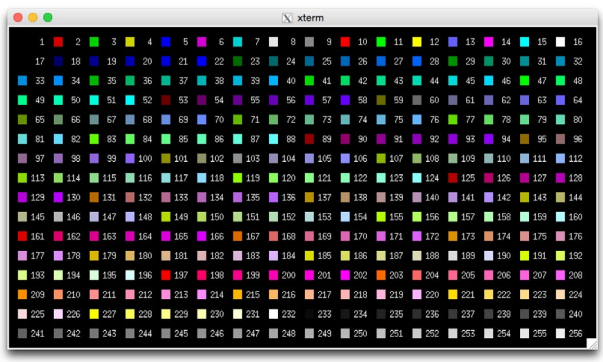
In that light, although Go is a productive language compared to C, the Go compiler's overt pedantry is a significant hindrance to trying out ideas with code, and getting one of those errors can be a real buzzkill. I still like writing Go code, but overall I fear that Go has sacrificed the values of fun, exploration, and knowledge-seeking in favor of the language designers' perceived political needs at their current place of employment.

Up From Below

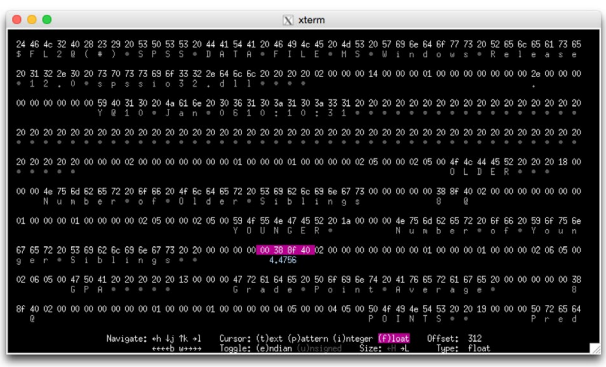
Despite my misgivings over the absence of Sloppy Go, and the waking nightmares I have about the Go gopher wearing my Peter Pan pajamas and murdering me in my sleep, on the whole I've been enjoying my initial experiences with the Go language. I was surprised at how idiot-proof it was to build things — you just type "go build" and almost instantly have a self-contained executable. This does make me wonder how things went so badly with make, makemaker, autoconf, aclocal, and the rest of the Texas Toolchain Massacre.

Termbox, by the way, is a fun library to work with. It gives you a key press handler and an API for putting colored characters at points; that's pretty much it. If you're feeling crushed beneath the twin behemoths of browser programming and scrum meetings, termbox is a great way to attempt to resuscitate your dying sense of worldly wonder and recapture your faded feelings of youth. I highly recommend it.

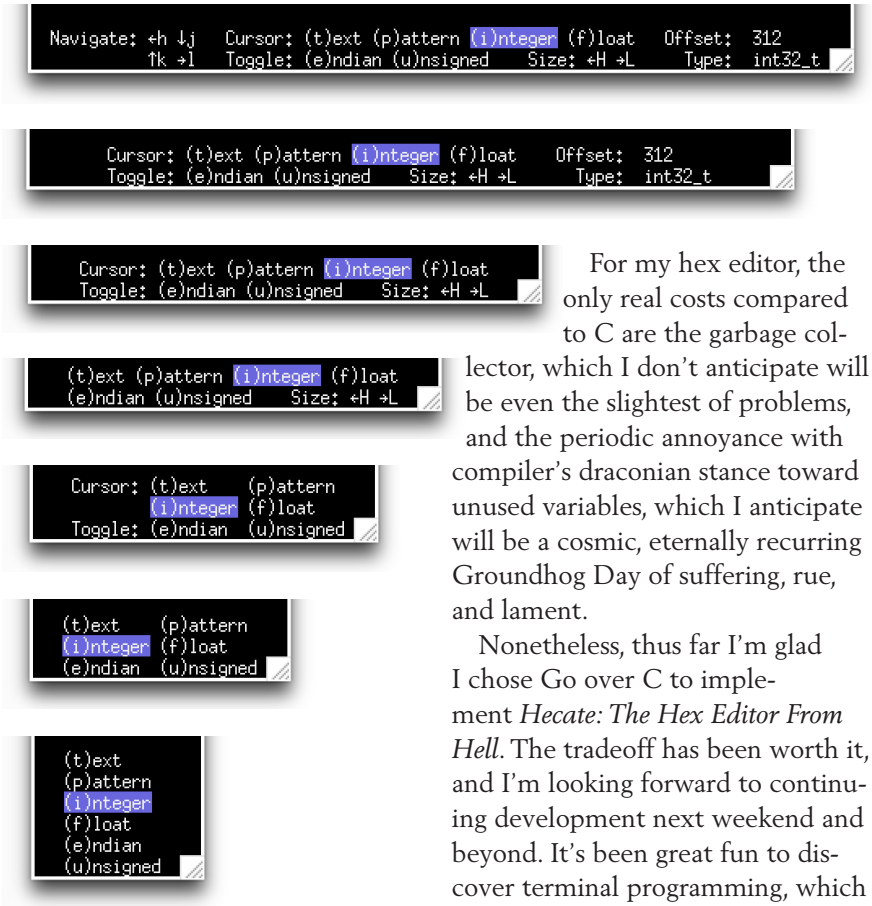
To get my initial groove on with termbox, I made a dumb program that displays all 256 terminal colors. It looks like this:



Once I figured out how to read a file, I had the beginnings of a hex viewer:



And check it out, responsive terminal design:



Go is productive enough that I've been enjoying implementing things from scratch like collapsible widgets and navigating a viewport. In order to do evil things like convert raw bytes to `floats`, I chose to use the "unsafe" package, which made me feel manly, powerful, and highly supportive of private gun ownership. Interfacing with C appears to be straightforward, though I feel like the compiler may want a criminal-background check and 30-day waiting period before letting me use it.

For my hex editor, the only real costs compared to C are the garbage collector, which I don't anticipate will be even the slightest of problems, and the periodic annoyance with compiler's draconian stance toward unused variables, which I anticipate will be a cosmic, eternally recurring Groundhog Day of suffering, rue, and lament.

Nonetheless, thus far I'm glad I chose Go over C to implement *Hecate: The Hex Editor From Hell*. The tradeoff has been worth it, and I'm looking forward to continuing development next weekend and beyond. It's been great fun to discover terminal programming, which is a welcome relief from worrying about embedded fonts and Retina displays and Apple Watch WebKit and whatnot. Who knows? Maybe one day I'll actually use Hecate to reverse-engineer another flag field in that binary file and proceed to take complete credit for it on my blog. ■

Evan Miller is the creator of Wizard, a next-generation statistics package for Mac.

Reprinted with permission of the original author.
First appeared in *hn.my/fourgo* (evanmiller.org)

Becoming Productive in Haskell

Coming From Python

By MATTHEW GRIFFITH

Somtime recently I became proficient enough in Haskell to be productive, and I wanted to capture some of my thoughts on the learning experience before it got too far away. I do most of my web prototyping in Haskell now, though I still regularly use and enjoy Python.

Data First

This is more of a thought on moving from a dynamic language to a static language, but in Haskell the structure of your data is mostly stated in data declarations and type signatures. In Python it's mostly implied by the code.

My first thought with a Haskell function is "What does the data look like? This function takes a ____ and returns a _____ ?", while in Python my first thought is "What does the code say?"

Thinking 'data first' improved my coding, even when coming back to Python. I more often recognize when the structure of my data changes for no real reason other than it was easy and I was very 'zoomed in' on the problem at the time.

Limiting changes in data structure also makes the code less complex and easier to understand.

The Readability

One of my main motivations for using Python is readability of code. Haskell originally looked ugly outside of what seemed to be carefully crafted examples. Pieces of it looked very clear, but were surrounded by flotsam and jetsam of nonsense. But it was also obviously powerful.

I definitely wanted to avoid 'clever' code that was powerful but confusing.

However, my ability to assess readability was in assessing other imperative languages. It was a bit like criticizing the readability of Mandarin as an English reader.

I found that Haskell is not 'clever but deceptive.' Of course you can write 'clever' code in Haskell, just like any language, but it's not the common case.

Actually, in Haskell that 'clever code' can only do so many clever things, as it's constrained by the type system. If it says it returns an Int, it will return an Int or fail to compile.

The more powerful and precise abstraction mechanisms that Haskell supplies just sometimes smell like the magic that I try to avoid in Python.

No, Really, the Readability

In the beginning, though, you kind of have to have faith that, yes, people do read it without any trouble and on a regular basis. Once over the hump, Haskell became very readable for me.

1. Type signatures. They're like getting a little summary at the top of a chapter of a book. With the added bonus that it's guaranteed to be true. Wouldn't that be great to have next time you try to learn another language?

   This is the chapter where Tommy goes to the market and buys a duck.

```
chapter :: Tommy -> Market ->
Duck
```

2. Composing functions out of other, smaller functions offers a big reduction in complexity. If they're named well, this allows you to write functions that are easy to read.
3. It's concise. You don't need a ton of code to express a powerful idea.

Infix Symbols and Noise

I also wanted to mention something about the infix functions that are common in Haskell code, too. Infix functions/operators are functions that go between two arguments instead of before. The classic example is + for addition.

In Haskell, we have a few infix symbols that are used regularly: $, <$>, <-, ->, etc., and they can create a sort of symbol-induced despair/anger in newcomers.

Don't despair! I know they reek of deceptive cleverness, but there are only a limited number of common ones. Once you know them you'll see they're useful and simple. I think there are maybe 5 infix symbols that I use on a regular basis.

That being said, I would say ignore the lens library in the beginning, as it has a ton of infix symbols. It's a very cool library, but you can get by just fine without it. Wait until you're comfortable creating medium-sized things in Haskell, and then approach it at your leisure.

A Whole New Vocabulary.

There are a lot of completely new words to learn when you learn Haskell. Things like *Functor* and *Monad*.

These words are going to feel heavier to learn for a few reasons. When starting to learn imperative programming, a lot of the new vocabulary has at least some familiarity. A loop brings to mind… well, loops. Race tracks, roller coasters, uhh….cereal.

We store memories by attaching them to previously made memories, so there is going to be a tendency for your brain to just shut off if too many of these new, heavy words show up in a sentence or paragraph. I had no associations with the word Functor, so it was hard to store.

My strategy in learning these words was to come up with my own name that made sense to me and mentally substitute it every time that heavy word came up. After a while, these made-up synonyms anchored me and I had no problem with the 'heavy word.'

For example: Functor.

In Haskell, this is something that can be mapped over. For example, a list is a Functor. This means there is a mapping function that takes another function and applies it to everything in the list and creates a list with the results.

```
map (+1) [1,2,3,4]
-- results in [2,3,4,5]
```

So, I started calling it *Mappable*. Mappable was easy for me to remember and was descriptive of what it did. A list is a Functor. A list is Mappable.

My Trusty Print Statement

In Python, my main development tool is the print statement/function.

In Haskell, my main development tool is the type system. It checks what I'd normally use print statements to check: what data a function is actually receiving or returning.

But! You can use Debug.Trace [hn.my/debugtrace] as a Python style print function without having to bother with Haskell's IO type. This can be very useful to get started. Though, once you get moving in Haskell, you probably won't use it as much as you think you would.

If you leave trace statements in your code after you're finished debugging…well, you will feel dirtier when you do that in Haskell than when you do it in Python.

The Best Monad Tutorial

was a Parsec tutorial.

When you hear about someone becoming productive in Haskell, it mostly involves a description of how they finally understood Monads. Well, damn, here it goes.

I needed to write a parser. I had something in Python, but due to my inexperience in writing parsers, the growing complexity of my code was slowing me down considerably.

So, I had some extra time. I thought maybe I should give it a go in Haskell.

I found the Youtube video, Parsing Stuff in Haskell, which explains how to create a JSON parser in Haskell using the Parsec library. [hn.my/parsinghaskell]

But it also inadvertently showed me how to use Monads and Applicatives as tools to create something I needed. It showed me how they function (har, har) and how they are related to each other.

After writing a parser with them, I began to understand other code that used them. I then started to understand their abstract nature… but that abstractness was a lesson for another day, not for starting out.

Also, Parsec provided enough structure that my inexperience in writing parsers did not really matter. In fact, as someone just learning Haskell, I was able to write a parser that was better in every measure (lower complexity, faster speed, better readability, easier extensibility), compared to what I could do as a programmer who has worked with Python for years but with no expertise in parsers.

The learning process was incredibly rewarding

Haskell is my main web prototyping language now for several reasons.

Well, reason 0 is I have the opportunity to choose what technology I use. I know that's a luxury.

1. I'm able to write a prototype faster, and that prototype is usually my production version.
2. I don't have to waste my time on trivial bugs.
3. The bugs I do encounter are generally more meaningful and lead me to understanding the problem more. Note: meaningful doesn't always mean harder.
4. Python taught me not to worry about speed that much. Haskell agreed with that but let me have it anyway.
5. Refactoring is a breeze. In Python, I always had a nagging feeling that I forgot to change some small part of your code that will be important later.
6. Excellent libraries. I feel that the basic guarantees of the Haskell language make the standard quality of libraries exceptionally high. Then there are libraries that were game-changers for me (Parsec and QuickCheck immediately come to mind, but there are others).
7. A helpful community.
8. Easy to scale up code to using many cores.
9. Haskell infrastructure is improving all the time. Last year, when GHC (which is the Haskell compiler) 7.8 came out, it doubled the performance of Warp, one of the prominent web servers that was already pretty fast.

And finally, I have to say that writing Haskell code comes with a deep level of satisfaction. It's more rewarding than most any coding experience I've had.

Where to start?

It can be tough to find a good starting point.

Here's how I would do it if I had to learn Haskell again.

First, reading at least Chapters 1 through 8 in Learn you a Haskell for Great Good. [learnyouahaskell.com]

Then!

1. **Write a small module that doesn't worry about IO.** Something like a Sudoku module that generates Sudoku puzzles. Don't worry about using a random number as a seed. Use Debug.Trace as your print statement to see what's going on. Generate a puzzle and Debug.Trace it to the screen. Create your own data types, and just use functions (i.e., no custom typeclasses).
2. **Turn that into a Website using either Scotty or Spock.** Keep it simple, a URL that shows a Sudoku puzzle. Then, a URL that produces JSON of a Sudoku puzzle.
3. **Mess around with real IO.** Try printing the puzzle to the terminal without Debug.Trace.
4. **Find incremental ways to add to it.** Design a file format for Sudoku puzzles and write a Parsec parser for it! Don't have the file format be JSON — make something up.

Good luck! ■

Matthew Griffith is a scientist, developer, and designer. He currently works as an Informatics Scientist at Evotec, handling automatic curation for large amounts of chemical data.

Reprinted with permission of the original author. First appeared in *hn.my/prohaskell* (mechanical-elephant.com)

Join the DuckDuckGo Open Source Community.

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

Featured IA: Regex Contributor: mintsoft
Get started at duckduckhack.com