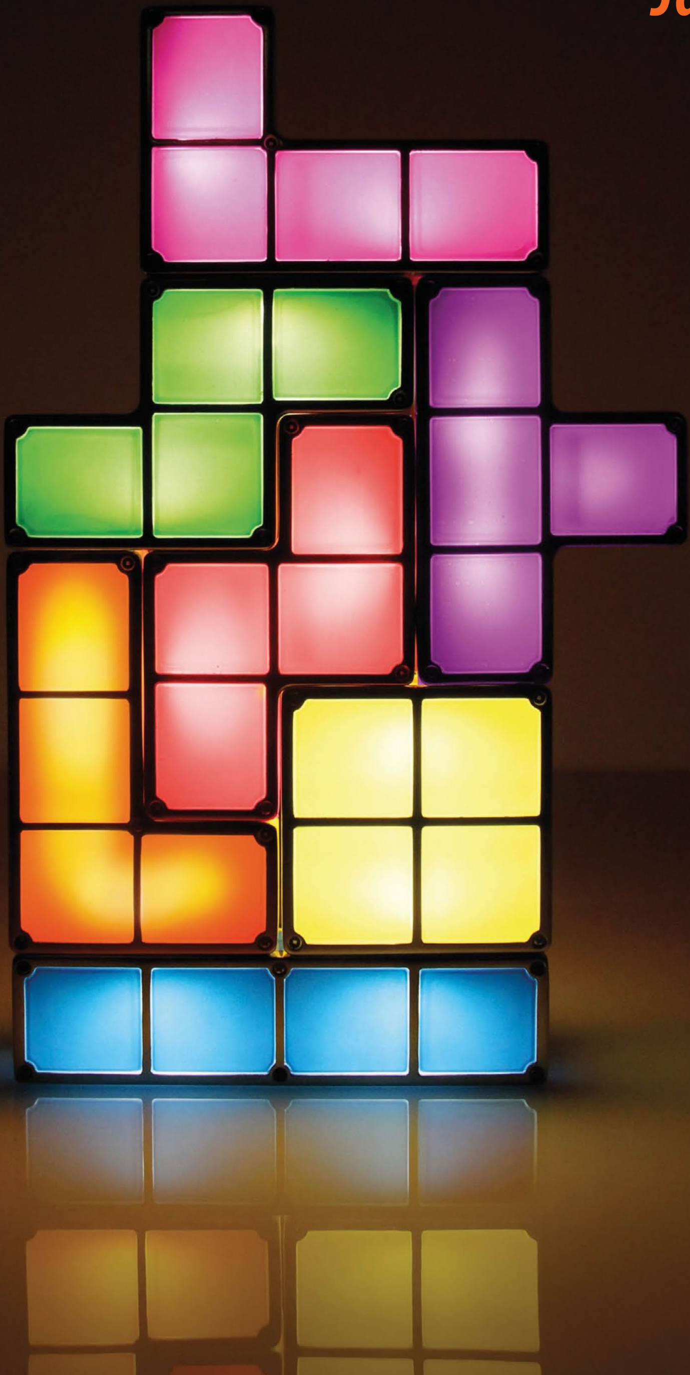
HACKERMONTHLY

Issue 58 March 2015

The Simple Proof of Tetris Lamp

Jack Morris



Curator
Lim Cheng Soon

Contributors
Jack Morris
Bob Nystrom
Dennis Felsing
Emily St.
Rasmus Borup Hansen
Francesca Krihely
Gabriel Gonzalez

Proofreader
Emily Griffin

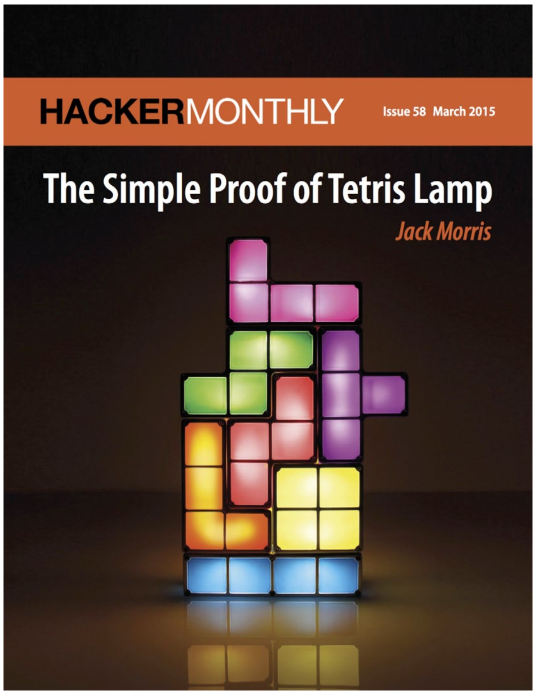
Printer
Blurb

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

Advertising
ads@hackermonthly.com

Contact
contact@hackermonthly.com

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

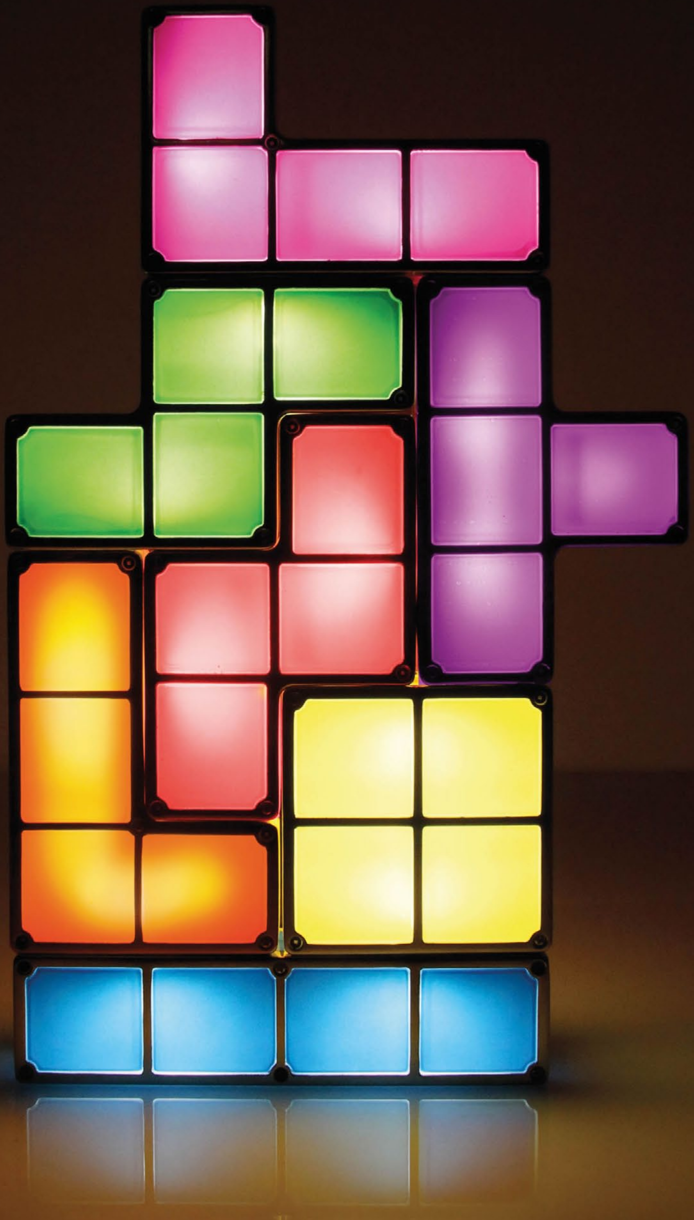


Contents

For links to Hacker News dicussions, visit *hackermonthly.com/issue-58*

The Simple Proof of the Tetris Lamp

By JACK MORRIS

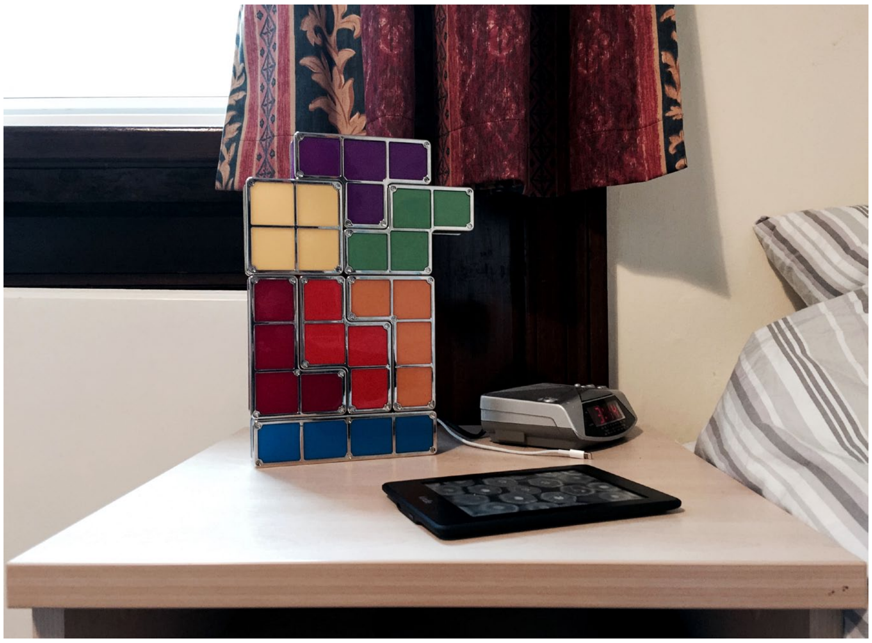


I received a Tetris lamp as a birthday present last year. It's a great little thing. You can move the individual Tetris pieces around to form whatever shape you wish, and once connected they all individually light up thanks to conducting strips around the edges of each segment.

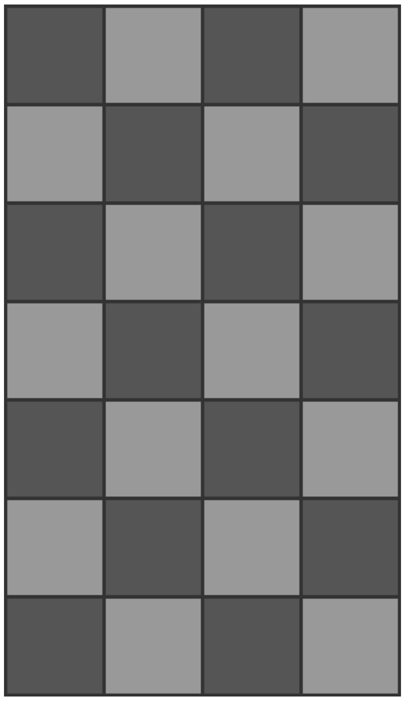
Leaving the obvious Tetris connection behind for a second though, one thing that's always irritated me is my inability to build the lamp into a clean rectangle. No matter how hard I tried I always ended up with a stray block sticking out of the side, and another missing on top, or some other irritating imperfect combination.

This irritation extended to many who have visited my room since the lamp became a fixture there. A friend of mine in particular spent an evening shuffling the pieces around into various positions, refusing to accept that there was someone out there with such a twisted mind that they'd happily design the pieces such that they didn't fit together in a clean way. Surely not.

He was inevitably unsuccessful in his endeavors, and I've since accepted that the lamp probably can't be constructed in such a way since the pieces just look like they won't fit together cleanly. That's not a particularly satisfactory conclusion however, and it definitely didn't quell my compulsive interest in the task.

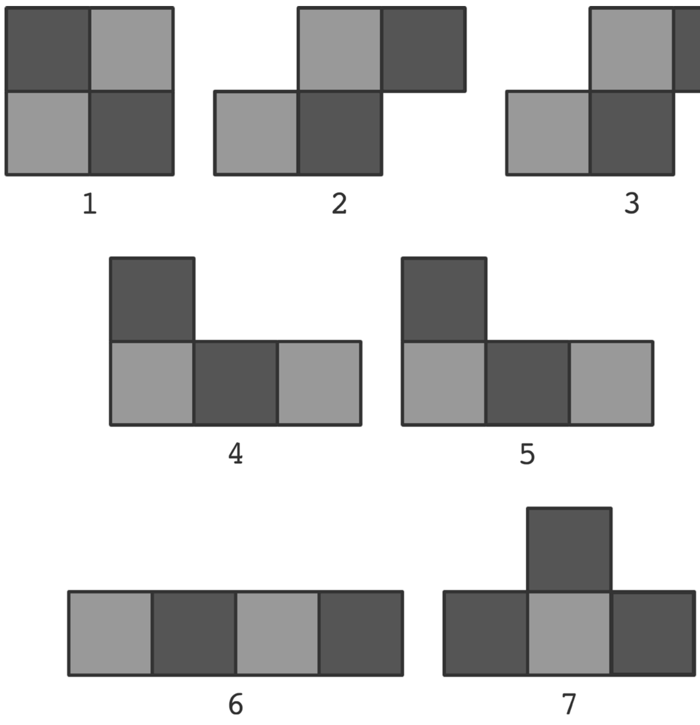

The lamp itself, in a standard irritating setup

However, while having drinks in my room last night, another friend (who hadn't been exposed to the lamp's tortuous attraction before) glanced at the construction on my desk, thought for a few minutes and exclaimed that he had a proof that it couldn't be formed into a rectangle. After hearing the details, the solution is so simple and elegant that I thought I'd share it here.



The lamp itself is composed of 7 individual pieces, containing a total of 28 squares. Therefore, assuming we can indeed form it into a rectangle, it would have to be 7x4 or 14x2 squares in size. I'm using the former case here simply because it's a more natural shape, however this proof applies equally as well to the latter. Now imagine that we label each of these squares with a color — either black or white — such that they form a checkerboard pattern as shown above. Notice that the number of black squares must be equal to the number of white, a property we'll exploit.

So that's 14 black squares, and 14 white. Looking at each of the pieces individually, the issue with our assumption quickly appears.



As shown above, for pieces 1-6, the number of black squares within the piece is equal to the number of white. Clearly which squares are black and which are white depends on the actual placement of the piece within the rectangle, but the shapes themselves dictate the count of each color (since adjacent squares must be different colors).

However, piece 7 disrupts the trend. Irrelevant of how it's located, it must be comprised of 3 squares of one color, and 1 of the other, a property that is purely down to its shape.

So, taking that into account along with the other 6 pieces, in total they're comprised of 13 squares of one color, and 15 of the other, with no assumptions about how they're located within the rectangle. Ah. We needed 14 of each, and since we've just shown that we can't get that, our original assumption is overturned and our proof is complete.

Conclusion

The proof itself is so simple that I'm slightly disappointed I didn't notice it myself sooner. However I'm glad that no more time will be wasted mindlessly moving around the pieces hoping for a breakthrough.

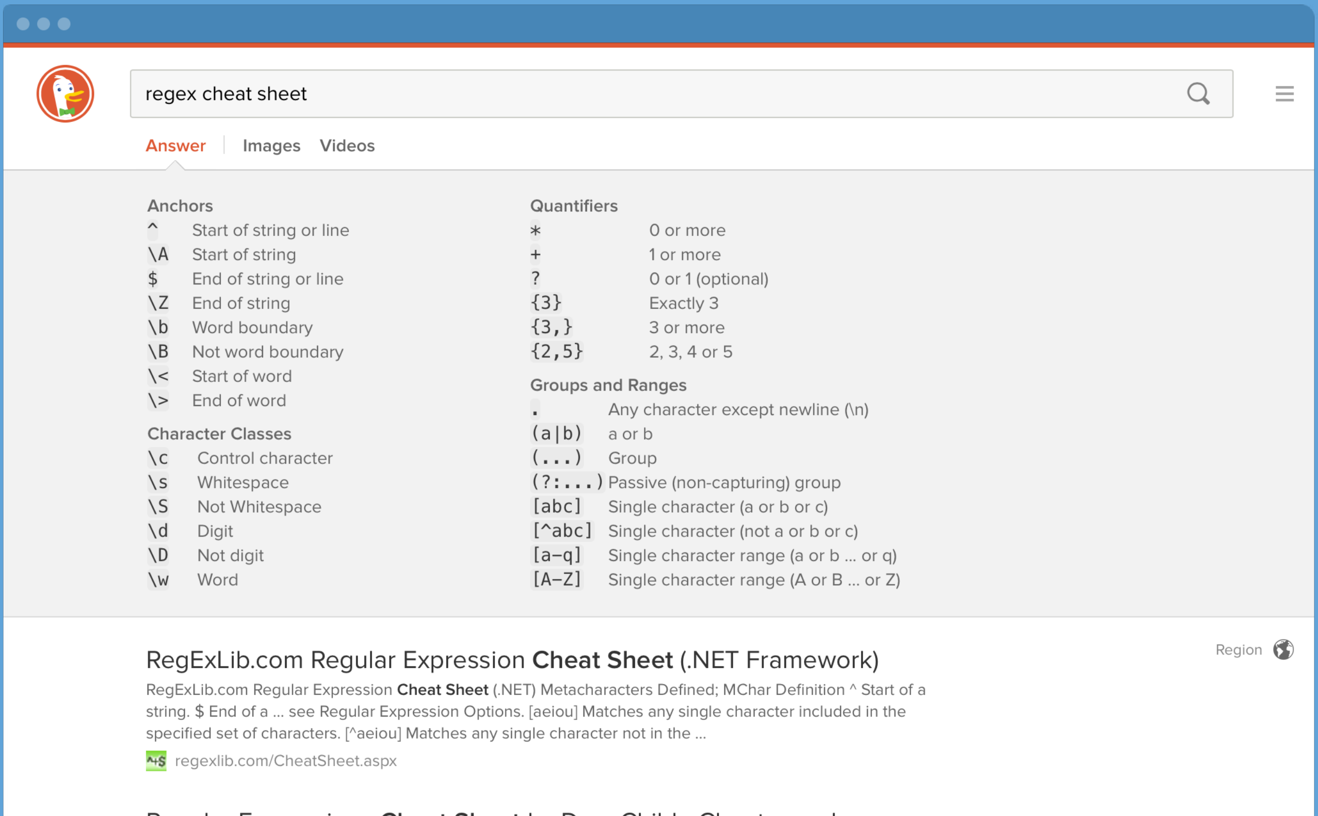
Maybe now I can shift my irritation from the lamp itself to whoever designed it to possess such a property. 🟧

Jack Morris is a 3rd year computer science undergraduate at the University of Cambridge, with an interest in iOS development and all things tech. Looking to move into industry next year he's currently seeking software development employment in London. Follow him on Twitter at *@jack_morris_* to be notified about future articles.

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



regex cheat sheet

Anchors

`^`	Start of string or line
`\A`	Start of string
`$`	End of string or line
`\Z`	End of string
`\b`	Word boundary
`\B`	Not word boundary
`\<`	Start of word
`\>`	End of word

Character Classes

`\c`	Control character
`\s`	Whitespace
`\S`	Not Whitespace
`\d`	Digit
`\D`	Not digit
`\w`	Word

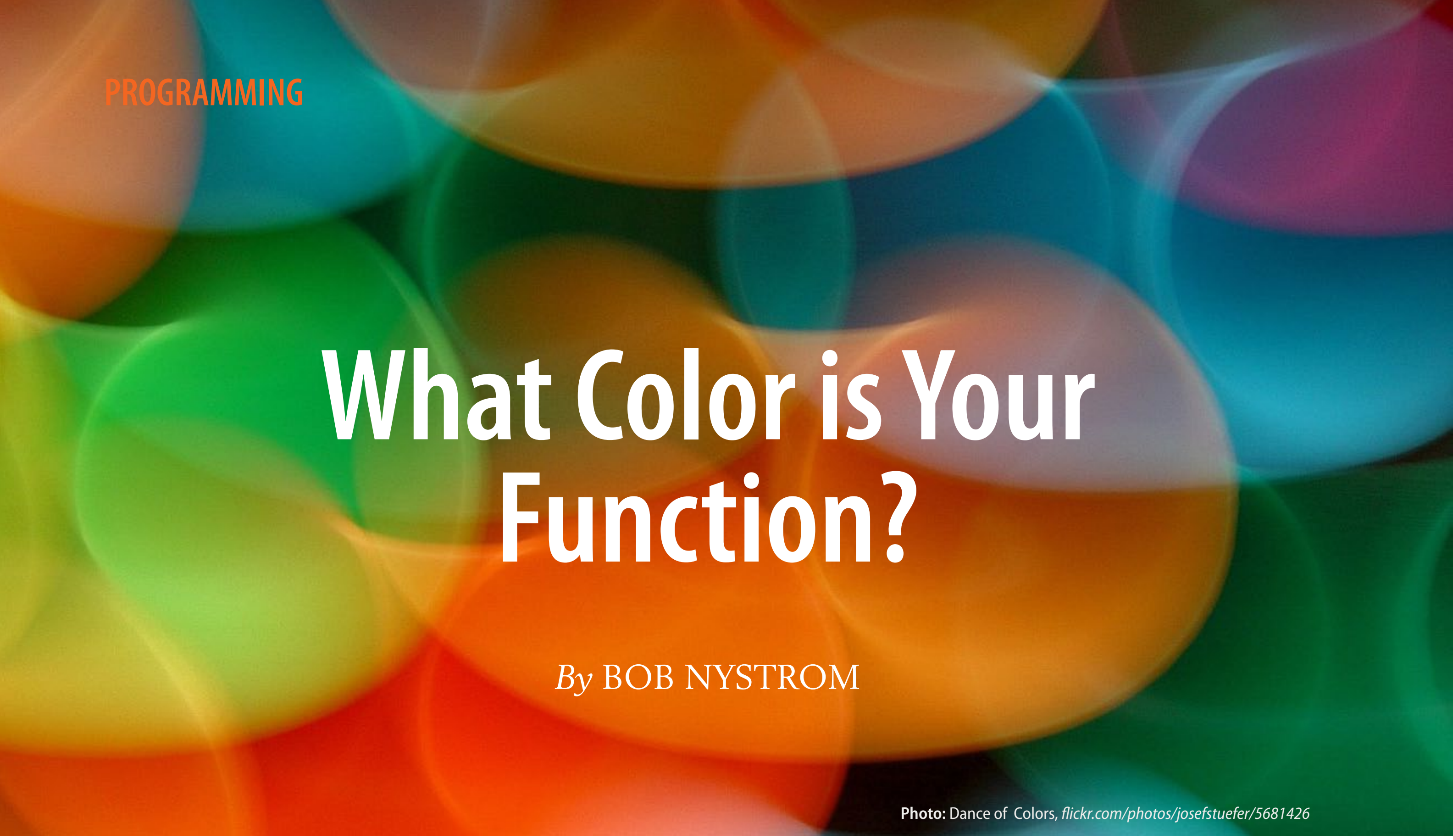
Quantifiers

`*`	0 or more
`+`	1 or more
`?`	0 or 1 (optional)
`{3}`	Exactly 3
`{3,}`	3 or more
`{2,5}`	2, 3, 4 or 5

Groups and Ranges

`.`	Any character except newline (\n)
`(a|b)`	a or b
`(...)`	Group
`(?:...)`	Passive (non-capturing) group
`[abc]`	Single character (a or b or c)
`[^abc]`	Single character (not a or b or c)
`[a-q]`	Single character range (a or b ... or q)
`[A-Z]`	Single character range (A or B ... or Z)

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



What Color is Your Function?

By BOB NYSTROM

Photo: Dance of Colors, *flickr.com/photos/josefstuefer/5681426*

I DON'T KNOW ABOUT you, but nothing gets me going in the morning quite like a good old fashioned programming language rant. It stirs the blood to see someone skewer one of those "blub" languages [hn.my/avg] the plebeians use, muddling through their day with it between furtive visits to StackOverflow.

(Meanwhile, you and I only use the most enlightened of languages. Chisel-sharp tools designed for the manicured hands of expert craftspeople such as ourselves.)

Of course, as the *author* of said screed, I run a risk. The language I mock could be one you like! Without realizing it, I could let have the rabble into my blog, pitchforks and torches at the ready, and my fool-hardy pamphlet could draw their ire!

To protect myself from the heat of those flames, and to avoid offending your possibly delicate sensibilities, instead, I'll rant about a language I just made up. A strawman whose sole purpose is to be set aflame.

I know, this seems pointless right? Trust me, by the end, we'll see whose face (or faces!) have been painted on his straw noggin.

A new language

Learning an entire new (crappy) language just for a bog post is a tall order, so let's say it's mostly similar to one you and I already know. We'll say it has syntax sort of like JS. Curly braces and semicolons. `if`, `while`, etc. The lingua franca of the programming grotto.

I'm picking JS not because that's what this article is about. It's just that it's the language you, statistical representation of the average reader, are most likely to be able grok. Voilà:

```
function thisIsAFunction() {
  return "It's awesome";
}
```

Because our strawman is a *modern* (shitty) language, we also have first-class functions. So you can make something like:

```
// Return a list containing all of the elements
// in collection that match predicate.
function filter(collection, predicate) {
  var result = [];
  for (var i = 0; i < collection.length; i++) {
    if (predicate(collection[i])) result.
push(collection[i]);
  }
  return result;
}
```

This is one of those higher-order functions, and, like the name implies, they are classy as all get out and super useful. You're probably used to them for mucking around with collections, but once you internalize the concept, you start using them damn near everywhere.

Maybe in your testing framework:

```
describe("An apple", function() {
  it("ain't no orange", function() {
    expect("Apple").not.toBe("Orange");
  });
});
```

Or when you need to parse some data:

```
tokens.match(Token.LEFT_BRACKET, function(token)
{
  // Parse a list literal...
  tokens.consume(Token.RIGHT_BRACKET);
});
```

So you go to town and write all sorts of awesome reusable libraries and applications passing around functions, calling functions, returning functions. Functapalooza.

What color is your function?

Except wait. Here's where our language gets screwy. It has this one peculiar feature:

❶ Every function has a color.

Each function — anonymous callback or regular named one — is either red or blue. Since my blog's code highlighter can't handle actual color, we'll say the syntax is like:

```
blue•function doSomethingAzure() {
  // This is a blue function...
}

red•function doSomethingCarnelian() {
  // This is a red function...
}
```

There are no colorless functions in the language. Want to make a function? Gotta pick a color. Them's the rules. And, actually, there are a couple more rules you have to follow, too:

❷ The way you call a function depends on its color.

Imagine a "blue call" syntax and a "red call" syntax. Something like:

```
doSomethingAzure(...)•blue;
doSomethingCarnelian()•red;
```

If you get it wrong — call a red function with `•blue` after the parentheses or vice versa — it does something bad. Dredge up some long-forgotten nightmare from your childhood like a clown with snakes for arms under your bed. That jumps out of your monitor and sucks out your vitreous humour.

Annoying rule, right? Oh, and one more:

❸ You can only call a red function from within another red function.

You can call a blue function from with a red one. This is kosher:

```
red•function doSomethingCarnelian() {
  doSomethingAzure()•blue;
}
```

But you can't go the other way. If you try to do this:

```
blue•function doSomethingAzure() {
  doSomethingCarnelian()•red;
}
```

Well, you're gonna get a visit from old Spidermouth the Night Clown.

This makes writing higher-order functions like our `filter()` example trickier. We have to pick a color for it, and that affects the colors of the functions we're allowed to pass to it. The obvious solution is to make `filter()` red. That way, it can take either red or blue functions and call them. But then we run into the next itchy spot in the hairshirt that is this language:

❹ Red functions are more painful to call.

For now, I won't precisely define "painful," but just imagine that the programmer has to jump through some kind of annoying hoops every time they call a red function. Maybe it's really verbose, or maybe you can't do it inside certain kinds of statements. Maybe you can only call them on line numbers that are prime.

What matters is that, if you decide to make a function red, everyone using your API will want to spit in your coffee and/or deposit some even less savory fluids in it.

The obvious solution then is to never use red functions. Just make everything blue and you're back to the sane world where all functions have the same color, which is equivalent to all of them having no color, which is equivalent to our language not being entirely stupid.

Alas, the sadistic language designers — and we all know all programming language designers are sadists, don't we? — jabbed one final thorn in our side:

❺ Some core library functions are red.
There are some functions built into the platform, functions that we need to use, that we are unable to write ourselves, that only come in red. At this point, a reasonable person might think the language hates us.

It's functional programming's fault!

You might be thinking that the problem here is we're trying to use higher-order functions. If we just stop flouncing around in all of that functional frippery and write normal blue collar first-order functions like God intended, we'd spare ourselves all the heartache.

If we only call blue functions, make our function blue. Otherwise, make it red. As long as we never make functions that accept functions, we don't have to worry about trying to be "polymorphic over function color" (polychromatic?) or any nonsense like that.

But, alas, higher order functions are just one example. This problem is pervasive any time we want to break our program down into separate functions that get reused.

For example, let's say we have a nice little blob of code that, I don't know, implements Dijkstra's algorithm over a graph representing how much your social network are crushing on each other. (I spent way too long trying to decide what such a result would even represent. Transitive undesirability?)

Later, you end up needing to use this same blob of code somewhere else. You do the natural thing and hoist it out into a separate function. You call it from the old place and your new code that uses it. But what color should it be? Obviously, you'll make it blue if you can, but what if it uses one of those nasty red-only core library functions?

What if the new place you want to call it is blue? You'll have to turn it red. Then you'll have to turn the function that calls it red. Ugh. No matter what, you'll have to think about color constantly. It will be the sand in your swimsuit on the beach vacation of development.

A colorful allegory

Of course, I'm not really talking about color here, am I? It's an allegory, a literary trick. The Sneetches isn't about stars on bellies, it's about race. By now, you may have an inkling of what color actually represents. If not, here's the big reveal:

Red functions are asynchronous ones.

If you're programming in JavaScript on Node.js, everytime you define a function that "returns" a value by invoking a callback, you just made a red function. Look back at that list of rules and see how my metaphor stacks up:

1. Synchronous functions return values, async ones do not and instead invoke callbacks.
2. Synchronous functions give their result as a return value, async functions give it by invoking a callback you pass to it.
3. You can't call an async function from a synchronous one because you won't be able to determine the result until the async one completes later.
4. Async functions don't compose in expressions because of the callbacks, have different error-handling, and can't be used with `try/catch` or inside a lot of other control flow statements.
5. Node's whole shtick is that the core libs are all asynchronous. (Though they did dial that back and start adding `___Sync()` versions of a lot of things.)

When people talk about "callback hell" they're talking about how annoying it is to have red functions in their language. When they create 4089 libraries for doing asynchronous programming, they're trying to cope at the library level with a problem that the language foisted onto them.

I promise the future is better

People in the Node community have realized that callbacks are a pain for a long time and have looked around for solutions. One technique that gets a bunch of people excited is promises, [hn.my/promise] which you may also know by their rapper name, "futures."

These are sort of a jacked up wrapper around a callback and an error handler. If you think of passing a callback and errorback to a function as a concept, a promise is basically a reification of that idea. It's a first-class object that represents an asynchronous operation.

I just jammed a bunch of fancy PL language in that paragraph so it probably sounds like a sweet deal, but it's basically snake oil. Promises do make async code a little easier to write. They compose a bit better, so rule #4 isn't quite so onerous.

But, honestly, it's like the difference between being punched in the gut versus punched in the privates. Less painful, yes, but I don't think anyone should really get thrilled about the value proposition.

You still can't use them with exception handling or other control flow statements. You still can't call a function that returns a future from synchronous code. (Well, you can, but if you do, the person who later maintains your code will invent a time machine, travel back in time to the moment that you did this and stab you in the face with a #2 pencil.)

You've still divided your entire world into asynchronous and synchronous halves and all of the misery that entails. So, even if your language features promises or futures, its face looks an awful lot like the one on my strawman.

(Yes, that means even Dart, [dartlang.org] the language I work on. That's why I'm so excited some of the team are experimenting with other concurrency models.)

I'm awaiting a solution

C# programmers are probably feeling pretty smug right now (a condition they've increasingly fallen prey to as Hejlsberg and company have piled sweet feature after sweet feature into the language). In C#, you can use the `await` keyword to invoke an asynchronous function.

This lets you make asynchronous calls just as easily as you can synchronous ones, with the tiny addition of a cute little keyword. You can nest `await` calls in expressions, use them in exception handling code, stuff them inside control flow. Go nuts. Make it rain `await` calls like they're dollars in the advance you got for your new rap album.

Async-await is nice, which is why we're adding it to Dart. It makes it a lot easier to write asynchronous code. You know a "but" is coming. It is. But… you still have divided the world in two. Those async functions are easier to write, but *they're still async functions.*

You've still got two colors. Async-await solves annoying rule #4: they make red functions not much worse to call than blue ones. But all of the other rules are still there:

1. Synchronous functions return values, async ones return `Task<T>` (or `Future<T>` in Dart) wrappers around the value.
2. Sync functions are just called, async ones need an `await`.
3. If you call an async function you've got this wrapper object when you actually want the `T`. You can't unwrap it unless you make your function async and await it. (But see below.)
4. Aside from a liberal garnish of `await`, we did at least fix this.
5. C#'s core library is actually older than async so I guess they never had this problem.

It is better. I will take async-await over bare callbacks or futures any day of the week. But we're lying to ourselves if we think all of our troubles are gone. As soon as you start trying to write higher-order functions or reuse code, you're right back to realizing color is still there, bleeding all over your codebase.

What language isn't colored?

So JS, Dart, C#, and Python have this problem. CoffeeScript and most other languages that compile to JS do, too (which is why Dart inherited it). I think even ClojureScript has this issue even though they've tried really hard to push against it with their `core.async` stuff.

Wanna know one that doesn't? Java. I know, right? How often do you get to say, "Yeah, Java is the one that really does this right"? But there you go. In their defense, they are actively trying to correct this oversight by moving to futures and async IO. It's like a race to the bottom.

C# also actually can avoid this problem, too. They opted in to having color. Before they added async-await and all of the `Task<T>` stuff, you just used regular sync API calls. Three more languages that don't have this problem: Go, Lua, and Ruby.

Any guess what they have in common?

Threads. Or, more precisely: *multiple independent callstacks that can be switched between*. It isn't strictly necessary for them to be operating system threads. Goroutines in Go, coroutines in Lua, and fibers in Ruby are perfectly adequate.

(That's why C# has that little caveat. You can avoid the pain of async in C# by using threads.)

Remembrance of operations past

The fundamental problem is: how do you pick up where you left off when an operation completes? You've built up some big callstack and then you call some IO operation. For performance, that operation uses the operating system's underlying asynchronous API. You cannot wait for it to complete because it won't. You have to return all the way back to your language's event loop and give the OS some time to spin before it will be done.

Once it is, you need to resume what you were doing. The usual way a language "remembers where it is" is the callstack. That tracks all of the functions that are currently being invoked and where the instruction pointer is in each one.

But to do async IO, you have to unwind discard the entire C callstack. Kind of a Catch-22. You can do super-fast IO, you just can't do anything with the result! Every language that has async IO in its bowels — or in the case of JS, the browser's event loop — copes with this in some way.

Node with its ever-marching-to-the-right callbacks stuffs all of those callframes in closures. When you do:

```
function makeSundae(callback) {
  scoopIceCream(function (iceCream) {
    warmUpCaramel(function (caramel) {
      callback(pourOnIceCream(iceCream, caramel));
    });
  });
}
```

Each of those function expressions closes over all of its surrounding context. That moves parameters like `iceCream` and `caramel` off the callstack and onto the heap. When the outer function returns and the callstack is trashed, it's cool. That data is still floating around the heap.

The problem is you have to manually reify every damn one of these steps. There's actually a name for this transformation: *continuation-passing style*. It was invented by language hackers in the "70s as an intermediate representation to use in the guts of their compilers. It"s a really bizarre way to represent code that happens to make some compiler optimizations easier to do.

No one ever for a second thought that a programmer would write actual code like that. And then Node came along and all of the sudden here we are pretending to be compiler back-ends. Where did we go wrong?

Note that promises and futures don't actually buy you anything, either. If you've used them, you know you're still hand-creating giant piles of function literals. You're just passing them to `.then()` instead of to the asynchronous function itself.

Awaiting a generated solution

Async-await does help. If you peel back your compiler's skull and see what it's doing when it hits an `await` call you'd see it actually doing the CPS-transform. That's why you need to use `await` in C#: it's a clue to the compiler to say, "break the function in half here." Everything after the `await` gets hoisted into a new function that it synthesizes on your behalf.

This is why async-await didn't need any runtime support in the .NET framework. The compiler compiles it away to a series of chained closures that it can already handle. (Interestingly, closures themselves also don't need runtime support. They get compiled to anonymous classes. In C#, closures really are a poor man's objects.)

You might be wondering when I'm going to bring up generators. Does your language have a `yield` keyword? Then it can do something very similar.

(In fact, I believe generators and async-await are isomorphic. I've got a bit of code floating around in some dark corner of my hard disc that implements a generator-style game loop using only async-await.)

Where was I? Oh, right. So with callbacks, promises, async-await, and generators, you ultimately end up taking your asynchronous function and smearing it out into a bunch of closures that live over in the heap.

Your function passes the outermost one into the runtime. When the event loop or IO operation is done, it invokes that function and you pick up where you left off. But that means everything above you also has to return. You still have to unwind the whole stack.

This is where the "red functions can only be called by red functions" rule comes from. You have to closurify the entire callstack all the way back to `main()` or the event handler.

Reified callstacks

But if you have threads (green- or OS-level), you don't need to do that. You can just suspend the entire thread and hop straight back to the OS or event loop *without having to return from all of those functions.*

Go is the language that does this most beautifully in my opinion. As soon as you do any IO operation, it just parks that goroutine and resumes any other ones that aren't blocked on IO.

If you look at the IO operations in the standard library, they seem synchronous. In other words, they just do work and then return a result when they are done. But it's not that they're synchronous in the sense that it would mean in JavaScript. Other Go code can run while one of these operations is pending. It's that Go has *eliminated the distinction between synchronous and asynchronous code.*

Concurrency in Go is a facet of how you choose to model your program, and not a color seared into each function in the standard library. This means all of the pain of the five rules I mentioned above is completely and totally eliminated.

So, the next time you start telling me about some new hot language and how awesome its concurrency story is because it has asynchronous APIs, now you'll know why I start grinding my teeth. Because it means you're right back to red functions and blue ones. ■

Robert Nystrom has programmed professionally for twenty years. He's worked on games, music applications, the web, and programming languages. The common thread, if there is one, is that he's most excited by making software that magnifies the creativity of others, whether that's other programmers using his code, or end users using his apps. Robert lives with his wife and two daughters in Seattle where you are most likely to find him cooking for his friends and plying them with good beer.

How I Start: Nim

By DENNIS FELSING

Nim is a young and exciting imperative programming language that is nearing its 1.0 release. My main motivation for using Nim [nim-lang.org] is its performance-to-productivity ratio and the joy of programming in Nim. In this guide I'm going to show you how I start a Nim project.

For this purpose we will write a small interpreter for the brainfuck language. [hn.my/brainfuck] While Nim is a practical language with many interesting features, brainfuck is the opposite: It's impractical to write in and its features consist of 8 single-character commands. Still, brainfuck is great for us, since its extreme simplicity makes it easy to write an interpreter for it. Later we will even write a high-performance compiler that transforms brainfuck programs into Nim at compile time. We will put all of this into a nimble package [hn.my/nimble] and publish it online. [hn.my/nimbf]

Installation

Installing Nim is straightforward, you can follow the official instructions. Binaries for Windows are provided. On other operating systems you can run the `build.sh` script to compile the generated C code, which should take less than a minute on a modern system.

This brings us to the first interesting fact about Nim: It compiles to C primarily (C++, ObjectiveC, and even JavaScript as well) and then uses the highly optimizing C compiler of your choice to generate the actual program. You get to benefit from the mature C ecosystem for free.

If you opt for bootstrapping the Nim compiler, [hn.my/nimc] which is written exclusively in Nim itself, you get to witness the compiler build itself with a few simple steps (in less than 2 minutes):

```
$ git clone https://github.com/Araq/Nim
$ cd Nim
$ git clone --depth 1
https://github.com/nim-lang/csources
$ cd csources && sh build.sh
$ cd ..
$ bin/nim c koch
$ ./koch boot -d:release
```

After you've finished the installation, you should add the `nim` binary to your path. If you use bash, this is what to do:

```
$ export PATH=$PATH:$your_install_dir/bin >>
~/.profile
$ source ~/.profile
$ nim
Nim Compiler Version 0.10.2 (2014-12-29) [Linux:
amd64]
Copyright (c) 2006-2014 by Andreas Rumpf
::

  nim command [options] [projectfile] [arguments]

Command:
  compile, c  compile project with default code
generator (C)
  doc         generate the documentation for
inputfile
  doc2        generate the documentation for the
whole project
  i           start Nim in interactive mode
(limited)
...
```

If `nim` reports its version and usage, we're good to continue. Now the modules from Nim's standard library are just an import away. All other packages can be retrieved with nimble, Nim's package manager. Let's follow the simple installation instructions. Again, for Windows a prebuilt archive is available, while building from source is quite comfortable as well:

```
$ git clone https://github.com/nim-lang/nimble
$ cd nimble
$ nim c -r src/nimble install
```

Nimble's binary directory wants to be added to your path as well:

```
$ export PATH=$PATH:$HOME/.nimble/bin >> ~/.pro-
file
$ source ~/.profile
$ nimble update
Downloading package list from https://github.
com/nim-lang/packages/raw/master/packages.json
```

Done.

Now we can browse the available nimble packages or search for them on the command line:

```
$ nimble search docopt
docopt:
  url:      git://github.com/docopt/docopt.nim
(git)
  tags:     commandline, arguments, parsing,
library
  description: Command-line args parser based on
Usage message
  license:  MIT
  website:  https://github.com/docopt/docopt.nim
```

Let's install this nice docopt library we found, maybe we'll need it later:

```
$ nimble install docopt
...
docopt installed successfully.
```

Notice how quickly the library is installed (less than 1 second for me). This is another nice effect of Nim. Basically the source code of the library is just downloaded, nothing resembling a shared library is compiled. Instead the library will simply be compiled statically into our program once we use it.

Project Setup

Now we're ready to get our project started:

```
$ mkdir brainfuck
$ cd brainfuck
```

First step: To get `Hello World` on the terminal, we create a `hello.nim` with the following content:

```
echo "Hello World"
```

We compile the code and run it, first in two separate steps:

```
$ nim c hello
$ ./hello
Hello World
```

Then in a single step, by instructing the Nim compiler to conveniently run the resulting binary immediately after creating it:

```
$ nim c -r hello
Hello World
```

Let's make our code do something slightly more complicated that should take a bit longer to run:

```
var x = 0
for i in 1 .. 100_000_000:
  inc x # increase x, this is a comment btw

echo "Hello World ", x
```

Now we're initializing the variable `x` to 0 and increasing it by 1 a whole 100 million times. Try to compile and run it again. Notice how long it takes to run now. Is Nim's performance that abysmal? Of course not, quite the opposite! We're just currently building the binary in full debug mode, adding checks for integer overflows, array out of bounds and much more, as well as not optimizing the binary at all. The `-d:release` option allows us to switch into release mode, giving us full speed:

```
$ nim c hello
$ time ./hello
Hello World 100000000
./hello  2.01s system 99% cpu 2.013 total
$ nim -d:release c hello
$ time ./hello
Hello World 100000000
./hello  0.00s system 74% cpu 0.002 total
```

That's a bit too fast, actually. The C compiler optimized away the entire `for` loop. Oops.

To start a new project `nimble init` can generate a basic package config file:

```
$ nimble init brainfuck
```

The newly created `brainfuck.nimble` should look like this:

```
[Package]
name          = "brainfuck"
version       = "0.1.0"
author        = "Anonymous"
description   = "New Nimble project for Nim"
license       = "BSD"

[Deps]
Requires: "nim >= 0.10.0"
```

Let's add the actual author, a description, as well as the requirement for docopt, as described in nimble's developers info. Most importantly, let's set the binary we want to create:

```
[Package]
name          = "brainfuck"
version       = "0.1.0"
author        = "The 'How I Start Nim' Team"
description   = "A brainfuck interpreter"
license       = "MIT"

bin           = "brainfuck"

[Deps]
Requires: "nim >= 0.10.0, docopt >= 0.1.0"
```

Since we have git installed already, we'll want to keep revisions of our source code and may want to publish them online at some point, let's initialize a git repository:

```
$ git init
$ git add hello.nim brainfuck.nimble .gitignore
```

Where I just initialized the `.gitignore` file to this:

```
nimcache/
*.swp
```

We tell git to ignore vim's swap files, as well as `nimcache` directories that contain the generated C code for our project. Check it out if you're curious how Nim compiles to C.

To see what nimble can do, let's initialize `brainfuck.nim`, our main program:

```
echo "Welcome to brainfuck"
```

We could compile it as we did before for `hello.nim`, but since we already set our package up to include the `brainfuck` binary, let's make `nimble` do the work:

```
$ nimble build
Looking for docopt (>= 0.1.0)...
Dependency already satisfied.
Building brainfuck/brainfuck using c backend...
...
$ ./brainfuck
Welcome to brainfuck
```

`nimble install` can be used to install the binary on our system, so that we can run it from anywhere:

```
$ nimble install
...
brainfuck installed successfully.
$ brainfuck
Welcome to brainfuck
```

This is great for when the program works, but `nimble build` actually does a release build for us. That takes a bit longer than a debug build, and leaves out the checks which are so important during development, so `nim c -r brainfuck` will be a better fit for now. Feel free to execute our program quite often during development to get a feeling for how everything works.

Coding

While programming, Nim's documentation comes in handy. If you don't know where to find what yet, there's a documentation index, in which you can search.

Let's start developing our interpreter by changing the `brainfuck.nim` file:

```
import os
```

First we import the os module, so that we can read command line arguments.

```
let code = if paramCount() > 0: readFile
paramStr(1)
           else: readAll stdin
```

`paramCount()` tells us about the number of command line arguments passed to the application. If we get a command line argument, we assume it's a filename, and read it in directly with `readFile paramStr(1)`. Otherwise we read everything from the standard input. In both cases, the result is stored in the `code` variable, which has been declared immutable with the `let` keyword.

To see if this works, we can `echo` the `code`:

```
echo code
```

And try it out:

```
$ nim c -r brainfuck
...
Welcome to brainfuck
I'm entering something here and it is printed
back later!
I'm entering something here and it is printed
back later!
```

After you've entered your "`code`" finish up with a new line and ctrl-d. Or you can pass in a filename, everything after `nim c -r brainfuck` is passed as command line arguments to the resulting binary:

```
$ nim c -r brainfuck .gitignore
...
Welcome to brainfuck
nimcache/
*.swp
On we go:
var
  tape = newSeq[char]()
  codePos = 0
  tapePos = 0
```

We declare a few variables that we'll need. We have to remember our current position in the `code` string (`codePos`) as well as on the `tape` (`tapePos`). Brainfuck works on an infinitely growing `tape`, which we represent as a `seq` of `char`s. Sequences are Nim's dynamic length arrays, other than with `newSeq` they can also be initialized using `var x = @[1, 2, 3]`.

Let's take a moment to appreciate that we don't have to specify the type of our variables, it is automatically inferred. If we wanted to be more explicit, we could do so:

```
var
  tape: seq[char] = newSeq[char]()
  codePos: int = 0
  tapePos: int = 0
```

Next we write a small procedure and call it immediately afterwards:

```
proc run(skip = false): bool =
  echo "codePos: ", codePos, " tapePos: ", tapePos

discard run()
```

There are a few things to note here:

- We pass a `skip` parameter, initialized to `false`.
- Obviously the parameter must be of type `bool`, then.
- The return type is `bool` as well, but we return nothing? Every result is initialized to binary 0 by default, meaning we return `false`.
- We can use the implicit `result` variable in every proc with a result and set `result = true`.
- Control flow can be changed by using `return true` to return immediately.
- We have to explicitly `discard` the returned `bool` value when calling `run()`. Otherwise the compiler complains with `brainfuck.nim(16, 3) Error: value of type 'bool' has to be discarded`. This is to prevent us from forgetting to handle the result.

Before we continue, let's think about the way brainfuck works. Some of this may look familiar if you've encountered Turing machines before. We have an input string `code` and a `tape` of `char`s that can grow infinitely in one direction. These are the 8 commands that can occur in the input string, every other character is ignored:

Op	Meaning	Nim equivalent
`>`	move right on `tape`	`inc tapePos`
`<`	move left on `tape`	`dec tapePos`
`+`	increment value on `tape`	`inc tape[tapePos]`
`-`	decrement value on `tape`	`dec tape[tapePos]`
`.`	output value on `tape`	`stdout.write tape[tapePos]`
`,`	input value to `tape`	`tape[tapePos] = stdin.readChar`
`[`	if value on `tape` is `\0`, jump forward to command after matching]	
`]`	if value on `tape` is not `\0`, jump back to command after matching [	

With this alone, brainfuck is one of the simplest Turing complete programming languages.

The first 6 commands can easily be converted into a case distinction in Nim:

```
proc run(skip = false): bool =
  case code[codePos]
  of '+': inc tape[tapePos]
  of '-': dec tape[tapePos]
  of '>': inc tapePos
  of '<': dec tapePos
  of '.': stdout.write tape[tapePos]
  of ',': tape[tapePos] = stdin.readChar
  else: discard
```

We are handling a single character from the input so far, let's make this a loop to handle them all:

```
proc run(skip = false): bool =
  while tapePos >= 0 and codePos < code.len:
    case code[codePos]
    of '+': inc tape[tapePos]
    of '-': dec tape[tapePos]
    of '>': inc tapePos
    of '<': dec tapePos
    of '.': stdout.write tape[tapePos]
    of ',': tape[tapePos] = stdin.readChar
    else: discard

    inc codePos
```

Let's try a simple program, like this:

```
$ echo ">+" | nim -r c brainfuck
Welcome to brainfuck
Traceback (most recent call last)
brainfuck.nim(26)         brainfuck
brainfuck.nim(16)         run
Error: unhandled exception: index out of bounds
[IndexError]
Error: execution of an external program failed
```

What a shocking result, our code crashes! What did we do wrong? The tape is supposed to grow infinitely, but we haven't increased its size at all! That's an easy fix right above the case:

```
    if tapePos >= tape.len:
      tape.add '\0'
```

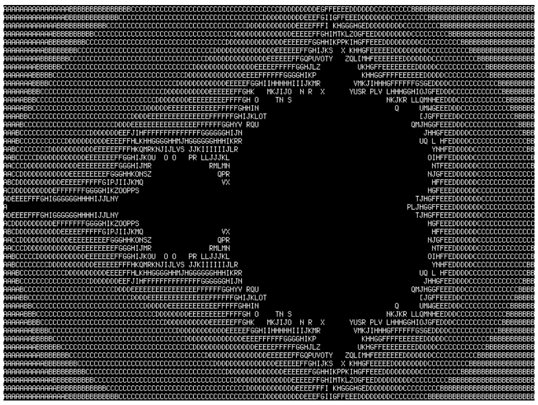
The last 2 commands, `[` and `]` form a simple loop. We can encode them into our code as well:

```
proc run(skip = false): bool =
  while tapePos >= 0 and codePos < code.len:
    if tapePos >= tape.len:
      tape.add '\0'

    if code[codePos] == '[':
      inc codePos
      let oldPos = codePos
      while run(tape[tapePos] == '\0'):
        codePos = oldPos
    elif code[codePos] == ']':
      return tape[tapePos] != '\0'
    elif not skip:
      case code[codePos]
      of '+': inc tape[tapePos]
      of '-': dec tape[tapePos]
      of '>': inc tapePos
      of '<': dec tapePos
      of '.': stdout.write tape[tapePos]
      of ',': tape[tapePos] = stdin.readChar
      else: discard

    inc codePos
```

If we encounter a `[` we recursively call the run function itself, looping until the corresponding `]` lands on a `tapePos` that doesn't have `\0` on the tape.

And that's it. We have a working brainfuck interpreter now. To test it, we create an `examples` directory containing these 3 files: `helloworld.b`, `rot13.b`, and `mandelbrot.b`.

```
$ nim -r c brainfuck examples/helloworld.b
Welcome to brainfuck
Hello World!
$ ./brainfuck examples/rot13.b
Welcome to brainfuck
You can enter anything here!
Lbh pna ragre nalguvat urer!
ctrl-d
$ ./brainfuck examples/mandelbrot.b
```



With the last one you will notice how slow our interpreter is. Compiling with `-d:release` gives a nice speedup, but still takes about 90 seconds on my machine to draw the Mandelbrot set. To achieve a great speedup, later on we will compile brainfuck to Nim instead of interpreting it. Nim's metaprogramming capabilities are perfect for this.

But let's keep it simple for now. Our interpreter is working, now we can turn our work into a reusable library. All we have to do is surround our code with a big `proc`:

```
proc interpret*(code: string) =
  var
    tape = newSeq[char]()
    codePos = 0
    tapePos = 0

  proc run(skip = false): bool =
    ...

  discard run()

when isMainModule:
  import os

  echo "Welcome to brainfuck"

  let code = if paramCount() > 0: readFile
paramStr(1)
             else: readAll stdin

  interpret code
```

Note that we also added a `*` to the proc, which indicates that it is exported and can be accessed from outside of our module. Everything else is hidden.

At the end of the file we still kept the `code` for our binary. `when isMainModule` ensures that this `code` is only compiled when this module is the main one. After a quick `nimble install` our `brainfuck` library can be used from anywhere on your system, just like this:

```
import brainfuck
interpret "++++++++[>++++[>++>+++>+++>+<<<<-
]>+>+>->>+[<]<-]>>.>---.+++++++..+++.>>.<-
.<.+++.------.--------.>>+.>++."
```

Looking good! At this point we could share the `code` with others already, but let's add some documentation first:

```
proc interpret*(code: string) =
  ## Interprets the brainfuck `code` string,
reading from stdin and writing to
  ## stdout.
  ...
```

`nim doc brainfuck` builds the documentation, which you can see online in its full glory. [hn.my/bfdoc]

Metaprogramming

As I said before, our interpreter is still pretty slow for the Mandelbrot program. Let's write a procedure that creates Nim code AST at compile time instead:

```
import macros

proc compile(code: string): PNimrodNode {.compi-
letime.} =
  var stmts = @[newStmtList()]

  template addStmt(text): stmt =
    stmts[stmts.high].add parseStmt(text)

  addStmt "var tape: array[1_000_000, char]"
  addStmt "var tapePos = 0"

  for c in code:
    case c
    of '+': addStmt "inc tape[tapePos]"
    of '-': addStmt "dec tape[tapePos]"
    of '>': addStmt "inc tapePos"
    of '<': addStmt "dec tapePos"
    of '.': addStmt "stdout.write tape[tapePos]"
    of ',': addStmt "tape[tapePos] = stdin.read-
Char"
    of '[': stmts.add newStmtList()
    of ']':
      var loop = newNimNode(nnkWhileStmt)
      loop.add parseExpr("tape[tapePos] !=
'\\0'")
      loop.add stmts.pop
      stmts[stmts.high].add loop
    else: discard

  result = stmts[0]
  echo result.repr
```

The template `addStmt` is just there to reduce boilerplate. We could also explicitly write the same operation at each position that currently uses `addStmt`. (And that's exactly what a template does!) `parseStmt` turns a piece of Nim code from a string into its corresponding AST, which we store in a list.

Most of the code is similar to the interpreter, except we're not executing the code now, but generating it, and adding it to a list of statements. `[` and `]` are more complicated: They get translated into a while loop surrounding the code in-between.

We're cheating a bit here, because we use a fixed size `tape` now and don't check for under- and overflows anymore. This is mainly for the sake of simplicity. To see what this code does, the last line, namely `echo result.repr` prints the Nim code we generated.

Try it out by calling it inside a `static` block, which forces execution at compile time:

```
static:
  discard compile "+>+[-]>,."
```

During compilation the generated code is printed:

```
var tape: array[1000000, char]
var codePos = 0
var tapePos = 0
inc tape[tapePos]
inc tapePos
inc tape[tapePos]
while tape[tapePos] != '\0':
  dec tape[tapePos]
inc tapePos
tape[tapePos] = stdin.readChar
stdout.write tape[tapePos]
```

Generally useful for writing macros is the `dumpTree` macro, which prints the AST of a piece of code (actual one, not as a string), for example:

```
import macros

dumpTree:
  while tape[tapePos] != '\0':
    inc tapePos
This shows us the following Tree:
StmtList
  WhileStmt
    Infix
      Ident !"!="
      BracketExpr
        Ident !"tape"
        Ident !"tapePos"
      CharLit 0
    StmtList
      Command
        Ident !"inc"
        Ident !"tapePos"
```

That's how I knew that we would need a `StmtList`, for example. When you do metaprogramming in Nim, it's generally a good idea to use `dumpTree` and print out the AST of the code you want to generate.

Macros can be used to insert the generated code into a program directly:

```
macro compileString*(code: string): stmt =
  ## Compiles the brainfuck `code` string into
  ## Nim code that reads from stdin
  ## and writes to stdout.
  compile code.strval

macro compileFile*(filename: string): stmt =
  ## Compiles the brainfuck code read from `file
  ## name` at compile time into Nim code that
  ## reads from stdin and writes to stdout.
  compile staticRead(filename.strval)
```

We can now compile the Mandelbrot program into Nim easily:

```
proc mandelbrot = compileFile "examples/
mandelbrot.b"

mandelbrot()
```

Compiling with full optimizations takes quite long now (about 4 seconds), because the Mandelbrot program is huge and GCC needs some time to optimize it. In return the program runs in just 1 second:

```
$ nim -d:release c brainfuck
$ ./brainfuck
```

Compiler settings

By default Nim compiles its intermediate C code with GCC, but clang usually compiles faster and may even yield more efficient code. It's always worth a try. To compile once with clang, use `nim -d:release --cc:clang c hello`. If you want to keep compiling `hello.nim` with clang, create a `hello.nim.cfg` file with the content `cc = clang`. To change the default backend compiler, edit `config/nim.cfg` in Nim's directory.

While we're talking about changing default compiler options: the Nim compiler is quite talky at times, which can be disabled by setting `hints = off` in the Nim compiler's `config/nim.cfg`. One of the more unexpected compiler warnings even warns you if you use `l` (lowercase `L`) as an identifier, because it may look similar to `1` (one):

```
a.nim(1, 4) Warning: 'l' should not be used
as an identifier; may look like '1' (one)
[SmallLshouldNotBeUsed]
```

If you're not a fan of this, a simple `warning[SmallLshouldNotBeUsed] = off` suffices to make the compiler shut up.

Another advantage of Nim is that we can use debuggers with C support, like GDB. Simply compile your program with `nim c --linedir:on --debuginfo c hello` and `gdb ./hello` can be used to debug your program.

Command line argument parsing

So far we've been parsing the command line argument by hand. Since we already installed the `docopt.nim` library before, we can use it now:

```
when isMainModule:
  import docopt, tables, strutils

  proc mandelbrot = compileFile("examples/
mandelbrot.b")

  let doc = """
brainfuck

Usage:
  brainfuck mandelbrot
  brainfuck interpret [<file.b>]
  brainfuck (-h | --help)
  brainfuck (-v | --version)

Options:
  -h --help     Show this screen.
  -v --version  Show version.
"""

  let args = docopt(doc, version = "brainfuck
1.0")

  if args["mandelbrot"]:
    mandelbrot()

  elif args["interpret"]:
    let code = if args["<file.b>"]:
readFile($args["<file.b>"])
               else: readAll stdin

    interpret(code)
```

The nice thing about docopt is that the documentation functions as the specification. Pretty simple to use:

```
$ nimble install
...
brainfuck installed successfully.
$ brainfuck -h
brainfuck

Usage:
  brainfuck mandelbrot
  brainfuck interpret [<file.b>]
  brainfuck (-h | --help)
  brainfuck (-v | --version)

Options:
  -h --help     Show this screen.
  -v --version  Show version.
$ brainfuck interpret examples/helloworld.b
Hello World!
```

Refactoring

Since our project is growing, we move the main source code into a `src` directory and add a `tests` directory, which we will soon need, resulting in a final directory structure like this:

```
$ tree
.
├── brainfuck.nimble
├── examples
│   ├── helloworld.b
│   ├── mandelbrot.b
│   └── rot13.b
├── license.txt
├── readme.md
├── src
│   └── brainfuck.nim
└── tests
    ├── all.nim
    ├── compile.nim
    ├── interpret.nim
    └── nim.cfg
```

This also requires us to change the nimble file:

```
srcDir = "src"
bin    = "brainfuck"
```

To improve reusability of our code, we turn to refactoring it. The main concern is that we always read from `stdin` and write to `stdout`.

Instead of accepting just a `code: string` as its parameter, we extend the `interpret` procedure to also receive an input and output stream. This uses the streams module that provides `FileStreams` and `StringStreams`:

```
## :Author: Dennis Felsing
##
## This module implements an interpreter for the
## brainfuck programming language as well as a
## compiler of brainfuck into efficient Nim code.
##
## Example:
##
## .. code:: nim
##   import brainfuck, streams
##
##   interpret("++++++++[>++++[>++>+++>+++>+<
## <<<-]>+>+>->>+[<]<-]>>.>---.+++++++..+++.>>.<-
## .<.+++.------.--------.>>+.>++.")
##   # Prints "Hello World!"
##
##   proc mandelbrot = compileFile("examples/
## mandelbrot.b")
##   mandelbrot() # Draws a mandelbrot set

import streams

proc interpret*(code: string; input, output:
Stream) =
  ## Interprets the brainfuck `code` string,
  ## reading from `input` and writing
  ## to `output`.
  ##
  ## Example:
  ##
  ## .. code:: nim
  ##   var inpStream = newStringStream("Hello
  ## World!\n")
  ##   var outStream = newFileStream(stdout)
  ##   interpret(readFile("examples/rot13.b"),
  ## inpStream, outStream)
```

I've also added some module wide documentation, including example code for how our library can be used. Take a look at the resulting documentation.

Most of the code stays the same, except the handling of brainfuck operations `.` and `,`, which now use `output` instead of `stdout` andinput instead of `stdin`:

```
      of '.': output.write tape[tapePos]
      of ',': tape[tapePos] = input.readCharEOF
```

What is this strange `readCharEOF` doing there instead of `readChar`? On many systems `EOF` (end of file) means `-1`. Our brainfuck programs actively use this. This means our brainfuck programs might actually not run on all systems. Meanwhile the streams module strives to be platform independent, so it returns a `0` if we have reached `EOF`. We use `readCharEOF` to convert this into a `-1` for brainfuck explicitly:

```
proc readCharEOF*(input: Stream): char =
  result = input.readChar
  if result == '\0': # Streams return 0 for EOF
    result = 255.chr # BF assumes EOF to be -1
```

At this point you may notice that the order of identifier declarations matters in Nim. If you declare `readCharEOF` below `interpret`, you cannot use it in `interpret`. I personally try to adhere to this, as it creates a hierarchy from simple code to more complex `code` in each module. If you still want to circumvent this, split declaration and definition of `readCharEOF` by adding this declaration above `interpret`:

```
proc readCharEOF*(input: Stream): char
```

The code to use the interpreter as conveniently as before is pretty simple:

```
proc interpret*(code, input: string): string =
  ## Interprets the brainfuck `code` string,
  ## reading from `input` and returning
  ## the result directly.
  var outStream = newStringStream()
  interpret(code, input.newStringStream, out-
Stream)
  result = outStream.data

proc interpret*(code: string) =
  ## Interprets the brainfuck `code` string,
  ## reading from stdin and writing to stdout.
  interpret(code, stdin.newFileStream, stdout.
newFileStream)
```

Now the `interpret` procedure can be used to return a string. This will be important for testing later:

```
let res = interpret(readFile("examples/
rot13.b"), "Hello World!\n")
interpret(readFile("examples/rot13.b")) # with
stdout
```

For the compiler, the cleanup is a bit more complicated. First we have to take the `input` and `output` as strings, so that the user of this proc can use any stream they want:

```
proc compile(code, input, output: string): PNim-
rodNode {.compiletime.} =
```

Two additional statements are necessary to initialize the input and output streams to the passed strings:

```
  addStmt "var inpStream = " & input
  addStmt "var outStream = " & output
```

Of course now we have to use `outStream` and `inpStream` instead of `stdout` and `stdin`, as well as `read-CharEOF` instead of `readChar`. Note that we can directly reuse the `readCharEOF` procedure from the interpreter, no need to duplicate `code`:

```
 of '.': addStmt "outStream.write tape[tapePos]"
 of ',': addStmt "tape[tapePos] = inpStream.
readCharEOF"
```

We also add a statement that will abort compilation with a nice error message if the user of our library uses it wrongly:

```
  addStmt """
    when not compiles(newStringStream()):
      static:
        quit("Error: Import the streams module
to compile brainfuck code", 1)
  """
```

To connect the new `compile` procedure to a `com-pileFile` macro that uses `stdout` and `stdin` again, we can write:

```
macro compileFile*(filename: string): stmt =
  compile(staticRead(filename.strval),
    "stdin.newFileStream", "stdout.new-
FileStream")
To read from an input string and write back to
an output string:
macro compileFile*(filename: string; input,
output: expr): stmt =
  result = compile(staticRead(filename.strval),
    "newStringStream(" & $input & ")", "newS-
tringStream()")
  result.add parseStmt($output & " = outStream.
data")
```

This unwieldy code allows us to write a compiled `rot13` procedure like this, connecting `input` string and `result` to the compiled program:

```
proc rot13(input: string): string =
  compileFile("../examples/rot13.b", input,
result)
echo rot13("Hello World!\n")
```

I did the same for `compileString` for convenience. You can check out the full code of `brainfuck.nim` on Github.

Conclusion

This is the end of our tour through the Nim ecosystem, I hope you enjoyed it and found it as interesting as it was for me to write it. ■

Dennis is an active contributor to the Nim language while working on his Master's thesis at KIT. There he worked on research developing a new method for Regression Verification and teaching programming paradigms (Haskell, lambda calculus, type inference, Prolog, Scala, etc.). He also develops and runs DDNet, a unique cooperative 2D game.

A Gentle Primer on Reverse Engineering

By EMILY ST.

OVER THE WEEKEND at Women Who Hack, [womenwhohack.org] I gave a short demonstration on reverse engineering. I wanted to show how "cracking" works, to give a better understanding of how programs work once they're compiled. It also serves my abiding interest in processors and other low-level stuff from the '80s.

My goal was to write a program which accepts a password and outputs whether the password is correct or not. Then I would compile the program to binary form (the way in which most programs are distributed) and attempt to alter the compiled program to accept any password. I did the demonstration on OS X, but the entire process uses open source tools from beginning to end, so you can easily do this on Windows (in a shell like Cygwin) or on Linux. If you want to follow along at home, I'm assuming an audience familiar with programming, in some form or another, but not much else.

Building a Program

I opened a terminal window and fired up my text editor (Vim) to create a new file called `program.c`. I wanted to write something that would be easy to understand and edit, and yet still could be compiled, so C seemed like a fine choice. My program wasn't doing anything that would've been strange in the year 1972.

First, I wrote a function for validating a password.

```
int is_valid(const char* password)
{
    if (strcmp(password, "poop") == 0) {
        return 1;
    } else {
        return 0;
    }
}
```

This function accepts a string and returns a `1` if the string is "poop" and `0` otherwise. I've chosen to call it `is_valid` to make it easier to find later. You'll understand what I mean a few sections down.

Now we need a bit of code to accept a string as input and call `is_valid` on it.

```
int main()
{
    char* input = malloc(256);3
    printf("Please input a word: ");
    scanf("%s", input);

    if (is_valid(input)) {
        printf("That's correct!\n");
    } else {
        printf("That's not correct!\n");
    }

    free(input);
    return 0;
}
```

This source code is likewise pretty standard. It prompts the user to type in a string and reads it in to a variable called `input`. Once that's done, it calls `is_valid` with that string. Depending on the result, it either prints “That’s correct!” or “That’s not correct!” and exits, returning control to the operating system. With a couple of “include” directives at the top, this is a fully functioning program.

Let’s build it! I saved the file `program.c` and used the command `gcc program.c -o program` to build it.

This outputs a file in the current directory called `program` which can be executed directly. Let’s run our program by typing `./program`. It’ll ask us to put in a word to check. We already know what to put in (“poop”), so let’s do that and make sure we see the result we expect.

```
Please input a word: poop
That's correct!
```

And if we run it again and type in the wrong word, we get the other possible result.

```
Please input a word: butts
That's not correct!
```

So far, so good.

A Deeper Look

There’s nothing special about this program that makes it different from your web browser or photo editor; it’s just a lot simpler. I can demonstrate this on my system with the `file` command. Trying it first on the program I just built, with the command `file program`, I see:

```
program: Mach-O 64-bit executable x86_64
```

This is the file format OS X uses to store programs. If this kind of file seems unfamiliar, the reason is that most applications are distributed as app bundles which are essentially folders holding the executable program itself and some ancillary resources. Again, with `file`, we can see this directly by running `file /Applications/Safari.app/Contents/MacOS/Safari`:

```
/Applications/Safari.app/Contents/MacOS/Safari:
Mach-O 64-bit executable x86_64
```

Let’s learn a little more about the binary we just built. We can’t open it in a text editor, or else we get garbage. Using a program called `hexdump` we can see the raw binary information (translated to hexadecimal) contained in the file. Let’s get a glimpse with `hexdump -C program | head -n 20`.

```
00000000  cf fa ed fe 07 00 00 01  03 00 00 80 02 00 00 00  |................|
00000010  10 00 00 00 10 05 00 00  85 00 20 00 00 00 00 00  |.......... .....|
00000020  19 00 00 00 48 00 00 00  5f 5f 50 41 47 45 5a 45  |....H...__PAGEZE|
00000030  52 4f 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |RO..............|
00000040  00 00 00 00 01 00 00 00  00 00 00 00 00 00 00 00  |................|
00000050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000060  00 00 00 00 00 00 00 00  19 00 00 00 28 02 00 00  |............(...|
00000070  5f 5f 54 45 58 54 00 00  00 00 00 00 00 00 00 00  |__TEXT..........|
00000080  00 00 00 00 01 00 00 00  00 10 00 00 00 00 00 00  |................|
00000090  00 00 00 00 00 00 00 00  00 10 00 00 00 00 00 00  |................|
000000a0  07 00 00 00 05 00 00 00  06 00 00 00 00 00 00 00  |................|
000000b0  5f 5f 74 65 78 74 00 00  00 00 00 00 00 00 00 00  |__text..........|
000000c0  5f 5f 54 45 58 54 00 00  00 00 00 00 00 00 00 00  |__TEXT..........|
000000d0  10 0e 00 00 01 00 00 00  e7 00 00 00 00 00 00 00  |................|
000000e0  10 0e 00 00 04 00 00 00  00 00 00 00 00 00 00 00  |................|
000000f0  00 04 00 80 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000100  5f 5f 73 74 75 62 73 00  00 00 00 00 00 00 00 00  |__stubs.........|
00000110  5f 5f 54 45 58 54 00 00  00 00 00 00 00 00 00 00  |__TEXT..........|
00000120  f8 0e 00 00 01 00 00 00  1e 00 00 00 00 00 00 00  |................|
00000130  f8 0e 00 00 01 00 00 00  00 00 00 00 00 00 00 00  |................|
```

The left column is the "offset," in hexadecimal (like line numbering, it tells us how many bytes into the file we are on a particular line). The middle two columns are the actual contents of the file itself, again in hexadecimal. The right column shows an ASCII equivalent for the file's contents, where possible. If you pipe the file's contents to `less` you can scan through and see mostly a lot of garbage and also a few familiar strings. If you're interested in knowing what pieces of text are embedded in a file, the program `strings` speeds this process up a great deal. In our case, it tells us:

```
poop
Please input a word:
That's correct!
That's not correct!
```

So clearly those strings are still floating around in the file. What's the rest of this stuff? Volumes of documentation exist out there on the Mach-O file format, but I don't want to bog down in the details. I have to level with you here — I honestly don't actually know much about it. Analogizing from other executable formats I've seen before, I know there's probably a header of some kind that helps the operating system know what kind of file this is and points out how the rest of the file is laid out. The rest of the file, incidentally, is made up of sections which may contain any of a number of things, including data (the strings in this case) built into the program; information on how to find code called from elsewhere in the system (imports, like our `printf` and `strcmp` functions, among others); and executable machine code.

Disassembling the Program

It's the machine code we're interested in now. This is the interesting part! Machine code is binary data, a long string of numbers which correspond to instructions the processor understands. When we run our program, the operating system looks at the file, lays it out in memory, finds the entry point, and starts feeding those instructions directly to the processor.

If you're used to scripted programming languages, this concept might seem a little odd, but it bears on what we're about to do to our binary. There's no interpreter going over things, checking stuff, making sure it makes sense, throwing exceptions for errors and ensuring they get handled. These instructions go right into the processor, and being a physical machine, it has no choice but to accept them and execute each one. This knowledge is very empowering because we have the final say over what these instructions are.

As you may know, the compiler `gcc` translated my source code I wrote earlier into machine language (and packaged it nicely in an executable file). This allows the operating system to execute it directly, but as another important consequence of this process, we also no longer need the source code. Most of the programs you run likely came as binary executables without source code at all. Others may have source code available, but they're distributed in binary form.

Whatever the case, let's imagine I lost the source code to `program` up above and can't remember it. Let's also imagine I can't even remember the password, and now my program holds hostage important secrets.

You might think I could run the binary through the `strings` utility, hoping the password gets printed out, and in this case, you'd be on the right track. Imagine if the program didn't have a single password built in and only accepted passwords whose letters were in alphabetical order or added up (in binary) a specific way. Without the source code, I couldn't scan to see which strings seem interesting, and I wouldn't have a clue what to type in.

But we don't need to lose heart because we already know that the program contains machine code, and since this machine code is meant to be fed directly to the processor, there's no chance it's been obfuscated or otherwise hidden. It's there, and it can't hide. If we knew how to read the machine code, there would be no need for the source code.

Machine code is hard for a human to read. There's a nice GNU utility called `objdump` which helps enormously in this respect. We'll use it to disassemble the binary. This process is called "disassembly" instead of "decompilation" because we can't get back the original source code; instead we can recover the names of the instructions encoded in machine code. It's not ideal, but we'll have to do our best. (Many people use a debugger to do this job, and there's a ton of benefits to doing so, like being able to watch instructions execute step by step, inspect values in memory, and so on, but a disassembly listing is simpler and less abstract.)

I looked up the documentation for `gobjdump` (as it's called on my system) and picked out some options that made sense for my purposes. I ended up running `gobjdump -S -l -C -F -t -w program | less` to get the disassembly. This is probably more than we'd care to know about our program's binary, much of it mysterious to me, but there's some very useful information here, too.

The Disassembly

I'll share at least what I can make of the disassembly. At the top of the listing is some general information. This symbol table is interesting. We can see the names of the functions I defined. If I had truly never seen the source code, I would at this point take an especial amount of interest in a function called `is_valid`, wouldn't I?

Immediately below this is a "Disassembly of section .text". I happen to know from past experience that the ".text" bit is a bit misleading for historical reasons; a ".text" section actually contains machine code! The leftmost column contains offsets (the place in the file where each instruction begins). The next column is the binary instructions themselves, represented in hexadecimal. After that are the names and parameters of each instruction (sometimes with a helpful little annotation left by `objdump`).

Of course, the very first thing I see is the instructions of the `is_valid` function.

```
Disassembly of section .text:

0000000100000e10  (File Offset: 0xe10):
   100000e10:   55                      push   %rbp
   100000e11:   48 89 e5                mov    %rsp,%rbp
   100000e14:   48 83 ec 10             sub    $0x10,%rsp
   100000e18:   48 89 7d f0             mov    %rdi,-0x10(%rbp)
   100000e1c:   48 8b 7d f0             mov    -0x10(%rbp),%rdi
   100000e20:   48 8d 35 33 01 00 00    lea    0x133(%rip),%rsi   # 100000f5a
(File Offset: 0xf5a)
   100000e27:   e8 e4 00 00 00          callq  100000f10  (File Offset: 0xf10)
   100000e2c:   3d 00 00 00 00          cmp    $0x0,%eax
   100000e31:   0f 85 0c 00 00 00       jne    100000e43  (File Offset: 0xe43)
   100000e37:   c7 45 fc 01 00 00 00    movl   $0x1,-0x4(%rbp)
   100000e3e:   e9 07 00 00 00          jmpq   100000e4a  (File Offset: 0xe4a)
   100000e43:   c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
   100000e4a:   8b 45 fc                mov    -0x4(%rbp),%eax
   100000e4d:   48 83 c4 10             add    $0x10,%rsp
   100000e51:   5d                      pop    %rbp
   100000e52:   c3                      retq
   100000e53:   66 66 66 66 2e 0f 1f 84 00 00 00 00 00  data16 data16 data16
nopw %cs:0x0(%rax,%rax,1)
```

This is super exciting because we're about to read assembly language. There are lots of books and sites on this subject, and my own understanding of assembly language is a bit rusty from years of disuse, but I know enough to get the gist. Let's break it down.

- The first three instructions (the first three lines, starting with `100000e10`) are a preamble that begin most functions in assembly language generated by a compiler. They're not important for us. (It saves the old frame pointer, gets a new frame pointer, and clears space on the stack for locals.)
- The next two instructions set up for our `strcmp` function. This looks a bit odd in assembly language compared to what we're used to. The `mov` instructions are shifting data from a location in memory to a register and vice versa. Because registers are involved, the disassembly wasn't able to hint very well what these values may be, but we can guess it's moving the strings to compare into place. I know this because of the calling convention for the function call (basically, set up the data and then make the call, which will know where to find the data); because the `%rbp` is the base register, which usually points to data; and because `-0x10(%rbp)` is a way of saying "look sixteen bytes earlier in memory than the address in the `%rbp` register."
- The `lea` and `callq` instructions load and call the `strcmp` function using the parameters we just moved in place. That function lives elsewhere in the system, so some magic happens here to transfer control of our program to that function.
- By the time we reach the `cmp` instruction, `strcmp` has done its thing and stored its result in the accumulator register `%eax`. By convention, return values usually live in `%eax`, so given that we're using a `cmp` ("compare"), and it's acting on `%eax` and `$0x0` (a zero), it's a safe bet we're checking to make sure `strcmp` returned zero. This instruction has the side effect of setting a flag in the processor called `ZF` to either `1` or `0`, depending on whether the comparison is true or not.
- The next instruction is `jne` which is short for "jump if not equal." It checks the `ZF` flag, and if it's zero, skips ahead twelve bytes (bypassing any instructions in the intervening space).
- That's followed by a `movl` and a `jmpq`. These instructions move a `1` into a location in memory and skip ahead another seven bytes. Look at the two-digit hexadecimal numbers to the left of these two instructions. They add up to twelve!
- Likewise, after these instructions, one other instruction moves the value `0` into the same location of memory and continues ahead. This instruction is exactly seven bytes long. So these jumps accomplish one of either two things: either the memory location `-0x4(%rbp)` is going to hold a `1` or a `0` by the time we get to the final `mov`. This is how assembly language does an `if` — a very interesting detail we'll return to.
- That last `mov` puts the value at `-0x4(%rbp)` (we just saw it's either a `1` or a `0`) into `%eax`, which we know is going to be the return value.
- Finally, the function undoes the work from the preamble and returns. (After that is some junk that's never executed.)

That was a lengthy explanation, so to sum up, we learned that the binary executable has a function called `is_valid`, and this function calls `strcmp` with some values and returns either a `1` or a `0` based on its return value. That's a pretty accurate picture based on what we know of the source code, so I'm pleased as punch!

Directly below the definition for this function is the `main` function. It's longer, but it's no more complex. It does the same basic tasks of moving values around, calling functions, inspecting the values, and branching based on this. Again, the values are difficult to get insight into because many registers are used, and there's a bit more setup. For the sake of brevity, I'll leave analyzing this function as an exercise for the reader (I promise it won't be on the test).

Breaking the Program

Remember, we don't have the slightest idea what the password is, and there's no good indication from the disassembly what it might be. Now that we have a good understanding of how the program works, we stand a good chance of modifying the program so that it believes any password is correct, which is the next best thing.

We can't modify this disassembly listing itself. It's output from `objdump` meant to help us understand the machine code (the stuff in the second column). We have to modify the `program` file itself by finding and changing those hexadecimal numbers somewhere in the file.

After looking over how both `is_valid` and `main` work, there are lots of opportunities to change the flow of the program to get the result we want, but we have to stay within a few rules. Notice how a lot of the instructions specify where other parts of the program are in terms of relative counts of bytes? That means that we can't change the number of bytes anywhere, or else we'd break all the symbol references, section locations, jumps, offsets, and so on. We also need to put in numbers which are valid processor instructions so that the program doesn't crash.

If this were your first program, I'd be forced to assume you wouldn't know what numbers mean what to the processor. Luckily, the disassembly gives us hints on how to attack it. Let's confine our possibilities (such as changing jump logic or overwriting instructions with dummy instructions) to only those we can exploit by using looking at this disassembly itself. There isn't a lot of variety here.

To me, one neat thing about `is_valid` stands out. Two of the lines are extremely similar: `movl $0x0,-0x4(%rbp)` and `movl $0x1,-0x4(%rbp)`. They do complementing things with the same memory location, use the same number of bytes (seven), involve the same setup, are near one another, and directly set up the return value for `is_valid`. This says to me the machine code for each instruction would be interchangeable, and by changing one or the other, we can directly change the return value for `is_valid` to whatever we want. It's a safe bet, with a function named that, we want it to return a `1`, but if we weren't sure, I could look ahead to the `main` function and see how its return value gets used later on.

In other words, we want to change `movl $0x0,-0x4(%rbp)` to be `movl $0x1,-0x4(%rbp)` so that no matter what, `is_valid` returns a one. The machine code for the instruction we have is `c7 45 fc 00 00 00 00`. Conveniently, the machine code for that precise instruction we want is just two lines above: `c7 45 fc 01 00 00 00`. The last challenge ahead is to find these bytes in the actual file and change them.

Where in the file are these bytes? Note that the listing says "File Offset: 0xe10" for the function `is_valid`. That's actually the count of bytes into the file we'd find the first instruction for this function (3648 bytes, in decimal), and the offset in the left column for the first instruction is "100000e10", so those offsets in the left column look like they tell where in the file each instruction's machine code is. The instruction we care about is at "100000e43", so it must be 3651 bytes into the file. We only need to change the fourth byte of the instruction, so we can add four to that count to get 3655 bytes.

Using `hexdump -C program | less` and scrolling ahead a bit, I find a line like this one:

```
00000e40  00 00 00 c7 45 fc 00
00  00 00 8b 45 fc 48 83 c4
|....E......E.H..|
```

Sure enough, there's the instruction, and the seventh byte on this line is the one we want to change. Patching a binary file from the command line is sort of difficult, but this command should do the trick:

```
printf '\x01' | dd of=program
bs=1 seek=3654 count=1
conv=notrunc
```

`dd` is writing to the file `program` (`of=program`), seeking by one byte at a time (`bs=1`), skipping ahead 3654 bytes past the first one to land on 3655 (`seek=3654`), changing only one byte (`count=1`), and not truncating the rest of the file (`conv=notrunc`).

Now I'll run the program the same way we did before (`./program`) and see if this worked.

```
Please input a word: butts
That's correct!
```

Success!

Conclusions

That's about it. It's a contrived example, and I knew it would work out before the end, but this is a great way to start learning how programs are compiled, how processors work, and how software cracking happens. The concepts here also apply themselves to understanding how many security exploits work on a mechanistic level. ■

Emily is a programmer for Simple living in Portland, Oregon, USA. She writes about tech, writing, and social justice on her personal site.

My Experience With Using cp To Copy 432 Million Files

By RASMUS BORUP HANSEN

I RECENTLY HAD TO copy a lot of files. Even though I have 20 years of experience with various Unix variants, I was still surprised by the behavior of `cp`, and I think my observations should be shared with the community.

The setup: An old Dell server (2 cores, 2 GB initially, 10 GB later, running Ubuntu Trusty) with a new Dell storage enclosure (MD 1200) containing 12 4 TB disks configured with RAID 6 for a total of 40 TB capacity allowing two drives to fail simultaneously. The server is used for our off-site backup, and the only thing it does is write stuff to the disks. We use rsnapshot for that, so most of the files have a high link count (30+).

One morning I was notified that a disk had failed. No big deal, this happens now and then. I called Dell and next day I had a replacement disk. While rebuilding, the replacement disk failed, and in the meantime another disk had also failed. Now Dell's support wisely suggested that I did not just replace the failed disks as the array may have been punctured. Apparently, and as I understand it, disks are only reported as failed when they have many bad blocks. If you're unlucky, you can lose data if 3 corresponding blocks on different disks become bad within a short time, so that the RAID controller does not have a chance to detect the failures, recalculate the data from the parity, and store it somewhere else. So even though only two drives flashed red, data might have been lost.

Having almost used up the capacity, we decided to order another storage enclosure, copy the files from the old one to the new one, and then get the old one into a trustworthy state and use it to extend the total capacity. Normally I'd have copied/moved the files at block-level (e.g., using dd or pvmove), but suspecting bad blocks, I went for a file-level copy because then I'd know which files contained the bad blocks. I browsed the net for other peoples' experience with copying many files and quickly decided that `cp` would do the job nicely.

Knowing that preserving the hardlinks would require bookkeeping of which files have already been copied, I also ordered 8 GB more RAM for the server and configured more swap space.

When the new hardware had arrived I started the copying, and at first it proceeded nicely at around 300 – 400 MB/s as measured with iotop. After a while the speed decreased considerably, because most of the time was spent creating hardlinks, and it takes time to ensure that the filesystem is always in a consistent state. We use XFS, and we were probably suffering for not disabling write barriers (which can be done when the RAID controller has a write cache with a trustworthy battery backup). As expected, the memory usage of the `cp` command increased steadily and was soon in the gigabytes.

After some days of copying, the first real surprise came: I noticed that the copying had stopped, and `cp` did not make any system calls at all according to strace. Reading the source code revealed that `cp` keeps track of which files have been copied in a hash table that now and then has to be resized to avoid too many collisions. When the RAM has been used up, this becomes a slow operation.

Trusting that resizing the hash table would eventually finish, the `cp` command was allowed to continue, and after a while it started copying again. It stopped again and resized the hash table a couple of times, each taking more and more time. Finally, after 10 days of copying and hash table resizing, the new file system used as many blocks and inodes as the old one according to `df`, but to my surprise the `cp` command didn't exit. Looking at the source again, I found that `cp` disassembles its hash table data structures nicely after copying (the "`forget_all`" call). Since the virtual size of the `cp` process was now more than 17 GB and the server only had 10 GB of RAM, it did a lot of swapping.

I had started `cp` with the "`-v`" option and piped its output (both `stdout` and `stderr`) to a `tee` command to capture the output in a (big!) logfile. This meant that somewhere the output from `cp` was buffered because my logfile ended in the middle of a line. Wanting the buffers to be flushed so that I had a complete logfile, I gave `cp` more than a day to finish disassembling its hash table, before giving up and killing the process.

As I write this, I'm running an "`ls -laR`" on both file systems to be sure that everything is copied. But unless the last missing part of the output from `cp` contained more error messages, it appears that only a single file had i/o errors (luckily we had another copy of it).

I know this is not going to happen right away, but it would be nice if `cp` somehow used a data structure where the bookkeeping could be done while waiting for i/o instead of piling up the bookkeeping. And unless old systems without working memory management must be supported, I don't see any harm in simply removing the call to the `forget_all` function towards the end of `cp.c`.

To summarize the lessons I learned:

- If you trust that your hardware and your filesystem are ok, use block level copying if you're copying an entire filesystem. It'll be faster, unless you have lots of free space on it. In any case it will require less memory.
- If you copy many files and want to preserve hardlinks, make sure you have enough memory if you copy at file level.
- Disassembling data structures nicely can take much more time than just tearing them down brutally when the process exits.
- The number of hard drives flashing red is not the same as the number of hard drives with bad blocks. With RAID 6 you don't need three drives flashing red to loose data, if you're unlucky. Fewer can do. The same will be true for RAID 5, where you can loose data with only one or no drive flashing red, if you're really unlucky. ■

Rasmus Borup Hansen lives in Copenhagen, Denmark. He works at Intomics, a company specialised in analysing biological big data for the pharmaceutical industry. He has degrees in Mathematics and Computer Science from the University of Copenhagen where he has previously worked at the Faculty of Science and the Department of Mathematical Sciences.

From Node.js to Go

By FRANCESCA KRIHELY

In looking back on the past year, the biggest difference we made in our tech stack was moving from Node.js to Go. After building the first iteration of Bowery in Node.js, we made the switch to Go in February 2014, and it has helped us speed up development and deployment.

Since then, our entire team has become dedicated Gophers. We've enjoyed using Go for its clear-cut standards and simpler workflow. To give you a peek into our Gopher hole, here's a few reasons we love working with Go.

Easy to write cross-platform code

One of the biggest reasons we switched to Go was because of how simple it is to compile code for different systems.

At Bowery, [bowery.io] we're building an app to help you and your team manage your development environments, and we need to efficiently support all operating systems: Linux, Windows, and OSX. In Go, you can define different files for different operating systems that implement functionality depending on the operating system. A good example came up when our teammate Larz was building Prompt, [hn.my/bprompt] a library for reading user input from the command line. Larz wanted to create a Go package that would implement a cross platform line-editing prompt. This was simple in Go: by creating different files for each OS, the Go compiler would build the file depending on the operating system.

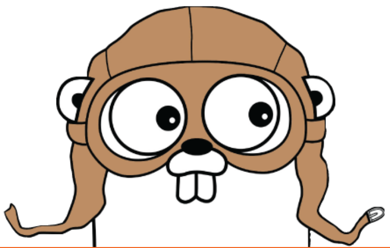
Compiling code for other systems is also simple: all you have to do is set an environment variable and you suddenly have a Windows binary that you built on a Linux system.

Faster deployment

Go is a compiled language, so distributing applications for use on multiple platforms is just easier. For us, this is important for deployment and testing, but also is an asset for our end users. With Go, build servers that run tests could easily just move on to production servers when they are ready. Go does not need any system dependencies, making it really simple to distribute. When it comes time to distribute out command line tools or other applications, our users don't need to worry about having Java, RVM, or NPM installed to run Bowery.

Concurrency primitives

When switching to Go, we realized the Node.js event loop wasn't everything. Node.js doesn't provide many concurrency primitives. The only thing running concurrently are I/O routines, timers, etc. You can't communicate across those routines, so it's challenging to build responsive systems on Node.js. With Go, you can run anything concurrently, and it provides channels to signal routines to do something or sends values across them to share data. Go also provides low level concurrency primitives like mutexes, wait groups, etc., which you could probably find on NPM. However, we find channels to be the deciding factor when dealing with concurrency and parallelization.



Integrated testing framework

With Node.js we had our choice of testing frameworks, but some worked better for front end, like Jasmine, and others were better for the backend, like Mocha. There are also other options like JSUnit and PhantomJS, and if you look on StackOverflow there are dozens of other frameworks suggested by users. In some worlds, choice is a good thing, but with Go, we liked the standardization of the testing framework. With Go, all the testing packages are built-in. If you need to write a new test suite, all you have to do is add the `_test.go` file to the same package as the software you are testing and it will run each time you execute `go test`.

Standard library

We love how you can write most software using only Go's standard library. With Node.js, we almost always had to include an external library; which increased deployment time and increased the potential instability that comes with using third party software. Being able to use just the standard library has enabled us to write code faster and safer.

Developer workflow tools are more powerful

With Node.js there's no real standardized workflow other than using NPM for packaging and script control. Other than that, the tooling is built by the community, which is great, but there are so many choices that the end result is everyone doing things differently.

A great example of workflow standardization in Go is the workspaces layout. You give up a lot of development freedom because you have to follow the workspaces layout, but it provides a lot of structure: you can keep all your Go source code and dependencies in one place. Within your workspace you have three root directories: `src` which holds source code for packages, `pkg` which holds the compiled packages, and `bin` which contains executable commands. It's a best practice to keep all of your source code and dependencies in a single workspace, making it standard across everyone's machine. The predictability is ideal when working on a team. We can go on anyone's machine to help and know for a fact our code is going to be in `$GOPATH/src/github.com/Bowery` rather than something like `$HOME/some/path/to/Bowery`. Similarly, gofmt formats everyone's code the same way. It's a huge relief that the superficial issues such as code organization and differences in code style just don't matter in Go. You can focus on fixing your problem and everything else is taken care of.

There's a ton of other reasons to like Go, and we're seeing more companies adopt it internally to power large, distributed applications. But overall, the Go team has discovered that developers can be more productive if you create standards and a set paradigm and others agree. For example, at MongoDB, the management applications team loves using Go for the "sensible, uniform development experience." At Soundcloud, they loved Go's strict formatting rules and "only one way to do things" philosophy. This means you spend less time in code review arguing about style and formatting and more time trying to solve your root problem. ■

Francesca is the CMO at Bowery. She's a big fan of open source technology and reads Hacker News for the articles.

Bowery is the Terminal that keeps your team in sync. With Bowery, your entire team can keep their runtimes up-to-date in the same way Github houses your code – by committing to a central location. Bowery hosts your environment and helps you share it with others so you can spend time focusing on what you do best, building your application.

Use Haskell for Shell Scripting

By GABRIEL GONZALEZ

RIGHT NOW DYNAMIC languages are popular in the scripting world, to the dismay of people who prefer statically typed languages for ease of maintenance.

Fortunately, Haskell is an excellent candidate for statically typed scripting for a few reasons:

- Haskell has lightweight syntax and very little boilerplate
- Haskell has global type inference, so all type annotations are optional
- You can type-check and interpret Haskell scripts very rapidly
- Haskell's function application syntax greatly resembles Bash
- However, Haskell has had a poor "out-of-the-box" experience for a while, mainly due to:
- Poor default types in the Prelude (specifically `String` and `FilePath`)
- Useful scripting utilities being spread over a large number of libraries
- Insufficient polish or attention to user experience (in my subjective opinion)

To solve this, I'm releasing the `turtle` library, [hn.my/turtle] which provides a slick and comprehensive interface for writing shell-like scripts in Haskell. I've also written a beginner-friendly tutorial targeted at people who don't know any Haskell.

Overview

`turtle` is a reimplementation of the Unix command line environment in Haskell. The best way to explain this is to show what a simple "turtle script" looks like:

```
#!/usr/bin/env runhaskell

{-# LANGUAGE OverloadedStrings #-}

import Turtle

main = do
    cd "/tmp"
    mkdir "test"
    output "test/foo" "Hello, world!"
-- Write "Hello, world!" to "test/foo"
    stdout (input "test/foo")
-- Stream "test/foo" to stdout
    rm "test/foo"
    rmdir "test"
    sleep 1
    die "Urk!"
```

If you make the above file executable, you can then run the program directly as a script:

```
$ chmod u+x example.hs
$ ./example.hs
Hello, world!
example.hs: user error (Urk!)
```

The `turtle` library renames a lot of existing Haskell utilities to match their Unix counterparts and places them under one import. This lets you reuse your shell scripting knowledge to get up and going quickly.

Shell compatibility

You can easily invoke an external process or shell command using `proc` or `shell`:

```
#!/usr/bin/env runhaskell

{-# LANGUAGE OverloadedStrings #-}

import Turtle

main = do
    mkdir "test"
    output "test/file.txt" "Hello!"
    proc "tar" ["czf", "test.tar.gz", "test"]
empty

-- or: shell "tar czf test.tar.gz test" empty
```

Even people unfamiliar with Haskell will probably understand what the above program does.

Portability

"`turtle` scripts" run on Windows, OS X and Linux. You can either compile scripts as native executables or interpret the scripts if you have the Haskell compiler installed.

Streaming

You can build or consume streaming sources. For example, here's how you print all descendants of the `/usr/lib` directory in constant memory:

```
#!/usr/bin/env runhaskell

{-# LANGUAGE OverloadedStrings #-}

import Turtle

main = view (lstree "/usr/lib")
... and here's how you count the number of
descendants:
#!/usr/bin/env runhaskell

{-# LANGUAGE OverloadedStrings #-}

import qualified Control.Foldl as Fold
import Turtle

main = do
    n <- fold (lstree "/usr/lib") Fold.length
    print n
```

... and here's how you count the number of lines in all descendant files:

```
#!/usr/bin/env runhaskell

{-# LANGUAGE OverloadedStrings #-}

import qualified Control.Foldl as Fold
import Turtle

descendantLines = do
    file <- lstree "/usr/lib"
    True <- liftIO (testfile file)
    input file

main = do
    n <- fold descendantLines Fold.length
    print n
```

Exception Safety

`turtle` ensures that all acquired resources are safely released in the face of exceptions. For example, if you acquire a temporary directory or file, `turtle` will ensure that it's safely deleted afterwards:

```
example = do
    dir <- using (mktempdir "/tmp" "test")
    liftIO (die "The temporary directory will
still be deleted!")
```

However, exception safety comes at a price. `turtle` forces you to consume all streams in their entirety so you can't lazily consume just the initial portion of a stream. This was a tradeoff I chose in order to keep the API as simple as possible.

Patterns

`turtle` supports `Patterns`, which are like improved regular expressions. Use `Patterns` as lightweight parsers to extract typed values from unstructured text:

```
$ ghci
>>> :set -XOverloadedStrings
>>> import Turtle
>>> data Pet = Cat | Dog deriving (Show)
>>> let pet = ("cat" *> return Cat) <|> ("dog" *> return
Dog) :: Pattern Pet
>>> match pet "dog"
>>> [Dog]
>>> match (pet `sepBy` ",") "cat,dog,cat"
[[Cat,Dog,Cat]]
```

You can also use `Patterns` as arguments to commands like sed, grep, or find, and they do the right thing:

```
>>> stdout (grep (prefix "c") "cat")
-- grep '^c'
cat
>>> stdout (grep (has ("c" <|> "d")) "dog")
-- grep 'cat\|dog'
dog
>>> stdout (sed (digit *> return "!") "ABC123")
-- sed 's/[[:digit:]]/!/g'
ABC!!!
```

Unlike many Haskell parsers, `Patterns` are fully backtracking, no exceptions.

Formatting

`turtle` supports typed `printf`-style string `formatting`:

```
>>> format ("I take "%d%" "%s%" arguments") 2 "typed"
"I take 2 typed arguments"
```

`turtle` even infers the number and types of arguments from the format string:

```
>>> :type format ("I take "%d%" "%s%" arguments")
format ("I take "%d%" "%s%" arguments") :: Text -> Int ->
Text
```

This uses a simplified version of the `Format` type from the `formatting` library. Credit to Chris Done for the great idea.

The reason I didn't reuse the `formatting` library was that I spent a lot of effort keeping the types as simple as possible to improve error messages and inferred types.

Learn more

`turtle` doesn't try to ambitiously reinvent `shell` scripting. Instead, `turtle` just strives to be a "better Bash." Embedding `shell` scripts in Haskell gives you the benefits of easy refactoring and basic sanity checking for your scripts.

You can find the `turtle` library on Hackage [hn.my/turtle] or Github. [hn.my/ghturtle] Also, `turtle` provides an extensive beginner-friendly tutorial targeted at people who don't know any Haskell at all. [hn.my/turtletut] ■

Gabriel Gonzalez builds analytics tools at Twitter and in his free time he does open source Haskell programming. He blogs about his work on *haskellforall.com* and you can reach him at *Gabriel439@gmail.com*

Metrics and monitoring for people who know what they want

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

Dashboards **StatsD** **Happiness**

Now with Grafana!

Why Hosted Graphite?

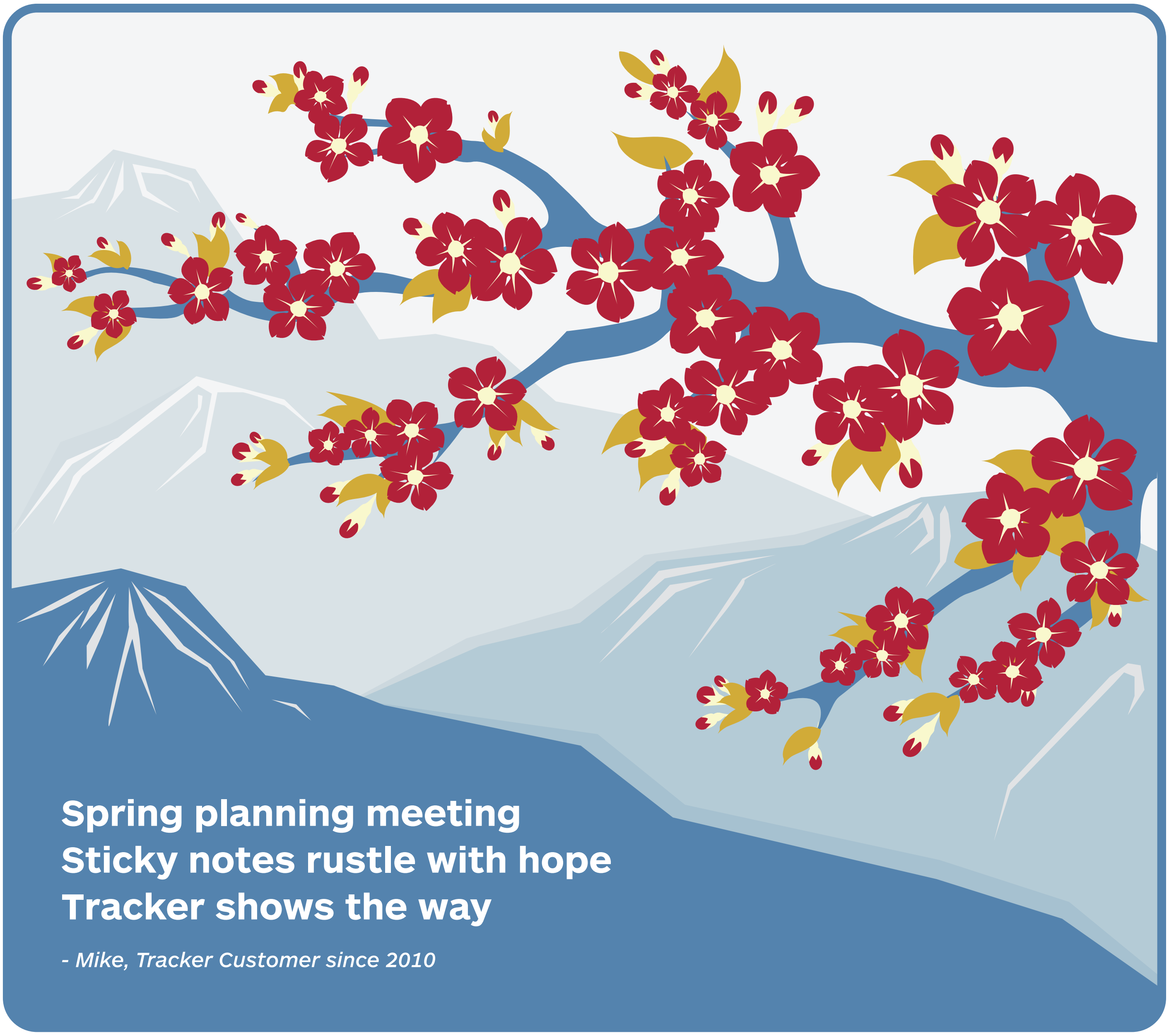
- **Hosted metrics and StatsD:** Metric aggregation without the setup headaches
- **High-resolution data:** See everything like some glorious mantis shrimp / eagle hybrid*
- **Flexible:** Lots of sample code, available on Heroku
- **Transparent pricing:** Pay for metrics, not data or servers
- **World-class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

Promo code: **HACKER**

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

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

HOSTEDGRAPHITE


Spring planning meeting
Sticky notes rustle with hope
Tracker shows the way
- Mike, Tracker Customer since 2010

Discover the newly redesigned Pivotal Tracker

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

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

Sign up for a free trial, no credit card required, at pivotaltracker.com.

PivotalTracker

Build better software, faster.