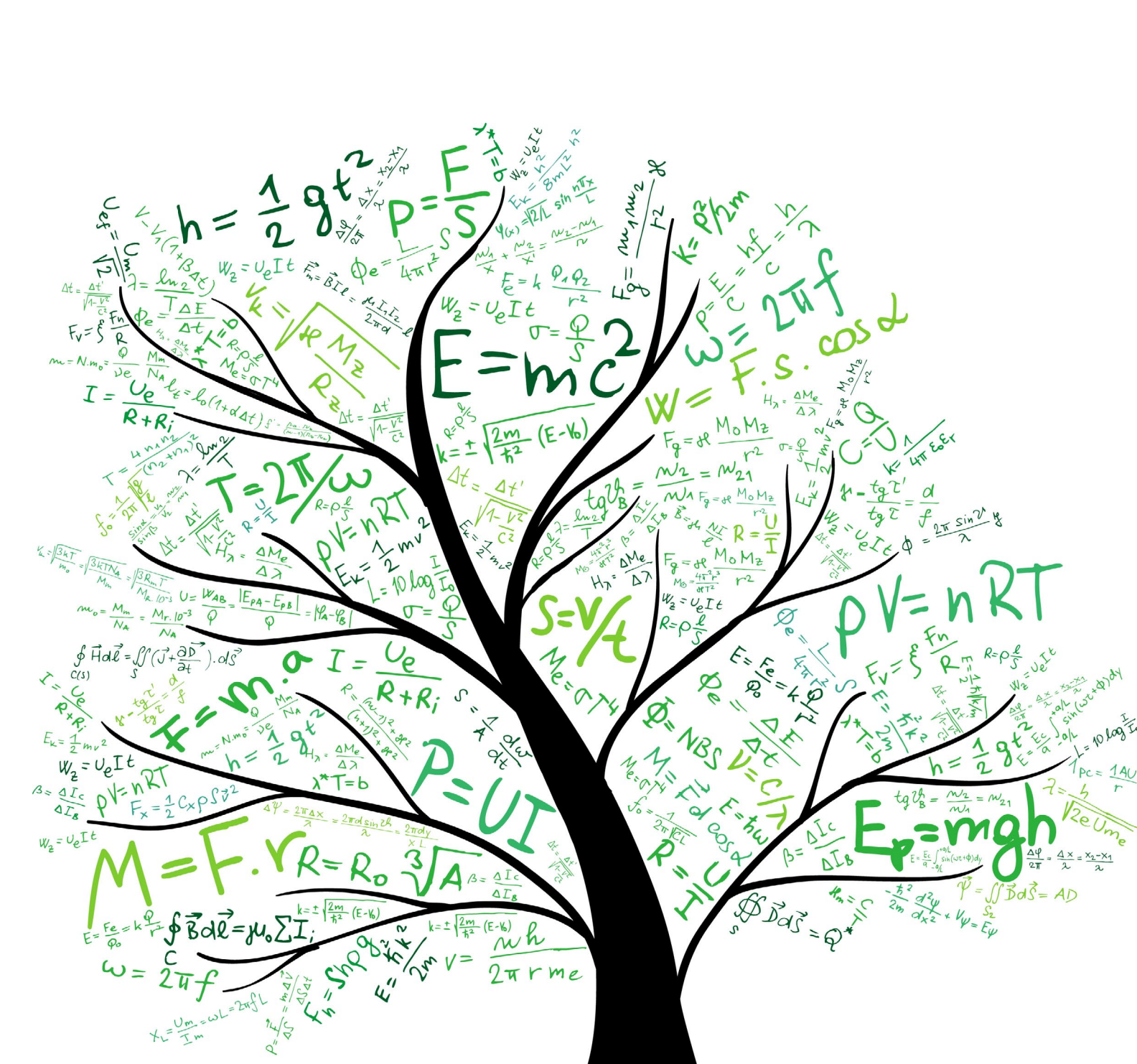


HACKERMONTHLY

Issue 20 January 2012

Google tracks you. We don't.

HARVEST

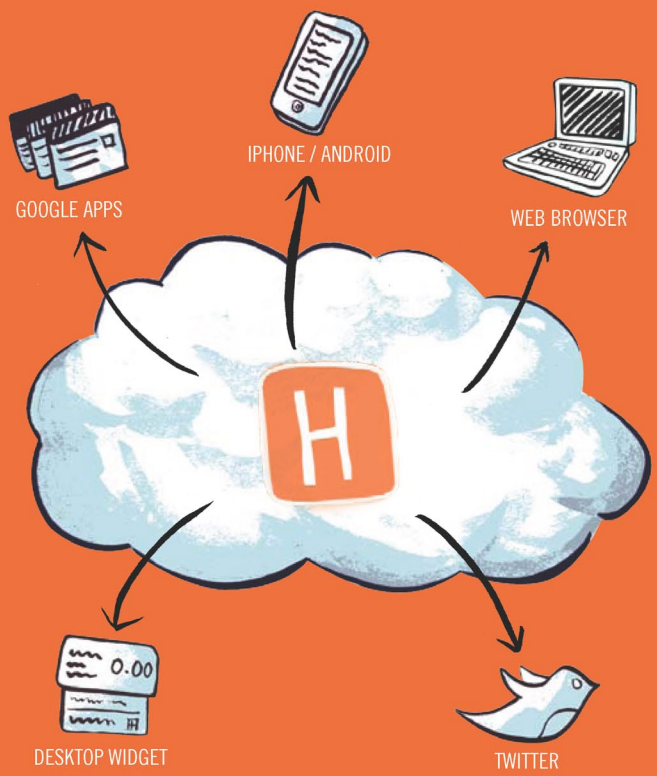
TIMESHEETS INVOICES REPORTS

Track time anywhere, and invoice your clients with ease.

Harvest is available wherever your work takes you. Whether you are working from home, on-site, or through a flight. Harvest keeps a handle on your billable time so you can invoice accurately. Visit us and learn more about how Harvest can help you work better today.

Why Harvest?

- Convenient and accessible, anywhere you go.
- Get paid twice as fast when you send web invoices.
- Trusted by small businesses in over 100 countries.
- Ability to tailor to your needs with full API.
- Fast and friendly customer support.



Learn more at **www.getHarvest.com/hackers**

Curator
Lim Cheng Soon

Contributors
Matt Might
Jason Cohen
Charlie Park
Edward Z. Yang
Chandra Patni
Alex MacCaw
Paul Stamatiou
Matthew Flickinger

Proofreader
Emily Griffin

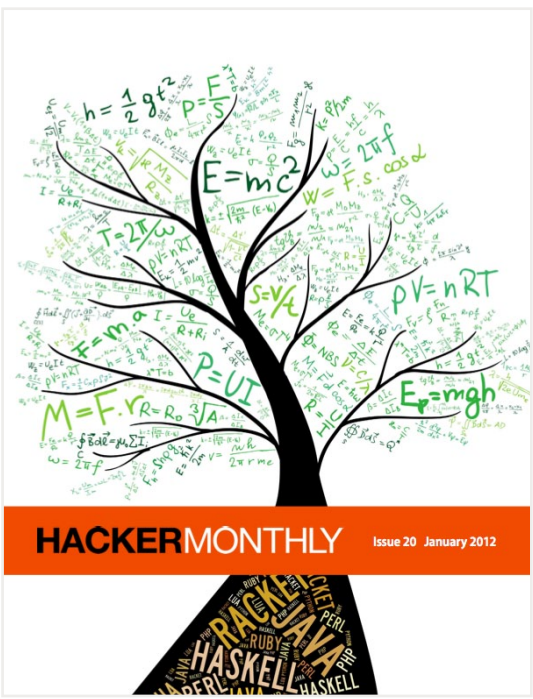
Printer
MagCloud

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

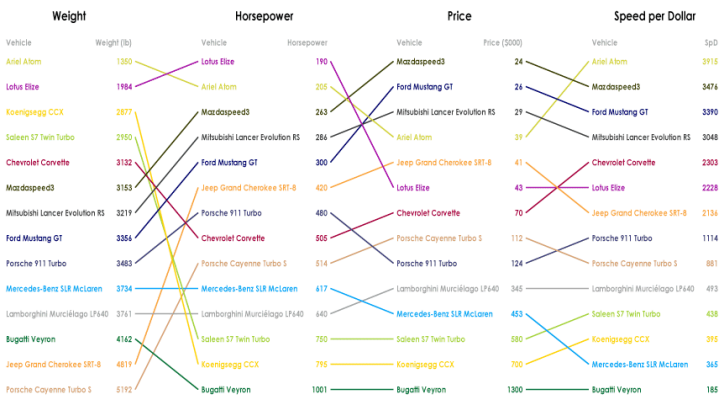
Advertising
ads@hackermonthly.com

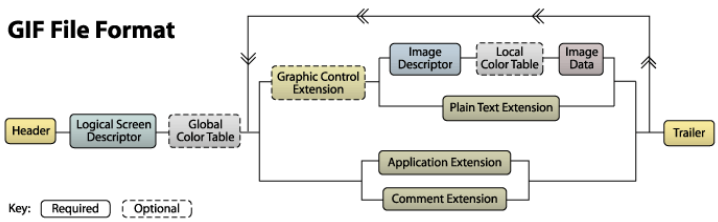
Contact
contact@hackermonthly.com

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



Contents





For links to Hacker News dicussions, visit *hackermonthly.com/issue-20*

Translating Math into Code

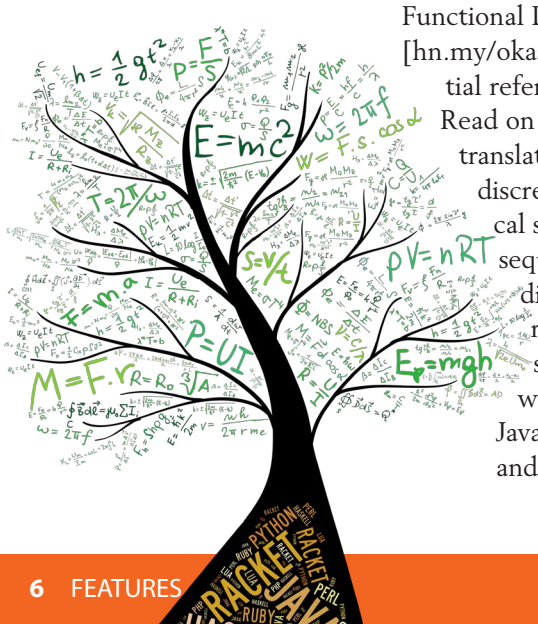
with Examples in Java, Racket, Haskell and Python

By MATT MIGHT

Discrete mathematical structures form the foundation of computer science.

These structures are so universal that most research papers in the theory of computation, programming languages, and formal methods present concepts in terms of discrete mathematics rather than code.

The underlying assumption is that the reader will know how to translate these structures into a faithful implementation as a working program.



A lack of material explaining this translation frustrates outsiders.

What deepens that frustration is that each language paradigm encodes discrete structures in a distinct way.

Many of the encodings are as immutable, purely functional data structures (even in imperative languages), a topic unfortunately omitted from many computer science curricula. Many standard libraries provide only mutable versions of these data structures, which instantly leads to pitfalls.

Okasaki's classic Purely Functional Data Structures [hn.my/okasaki] is an essential reference.

Read on for my guide to translating the common discrete mathematical structures — sets, sequences, functions, disjoint unions, relations and syntax — into working code in Java, Python, Racket, and Haskell.

Caution: Math has no side effects

The fatal mistake newcomers make when translating math into code is using mutable data structures where only an immutable structure was correct.

Mathematics has no side effects.

Math cannot modify the value of a variable, either global or local. It cannot mutate an element in an array. And a mathematical function always returns the same value for the same input.

The literal rendering of mathematics into code cannot contain side effects.

Mathematics is a purely functional language.

Of course, once the constraints on an implementation are understood, it's usually possible to exchange immutable data structures for mutable ones in key places to achieve performance savings.

But, for the purposes of prototyping, it's always best to start with a direct, purely functional implementation.

Sets and power sets

The rendering of a set as code will usually be a type, a collection backed by a balanced tree or a hash map, or a predicate.

In mathematics, a set is an unordered collection of elements.

The empty set, $\varnothing$, is a special set containing no elements.

The syntax for literal sets is curly braces: {}. For example, the set {1,2,3} is the set containing 1, 2 and 3.

The relationship $x \in S$ declares that the value x is a member of the set S.

Sets as types

Infinite sets tend to be encoded as types. (Of course, some finite sets are encoded as types too.)

In some cases, a set X is defined as a subset of another set Y:

$$X \subset Y.$$

This subset relationship could be represented as inheritance in a language like Java or Python, if these sets are meant to be types:

```
class X extends Y { ... }
```

When a set X is defined to be the power set of another set Y:

$$X = P(Y) = 2^Y,$$

then X and Y will be types, and members of X will be collections.

Sets as collections

When a set's contents are computed at run-time, then it will often be a sorted collection backed by a structure like a red-black tree.

It's not hard to implement a purely functional, sorted (but unbalanced) search tree in Java:

```
interface Ordered <T> {
  public boolean isLessThan(T that) ;
}

abstract class SortedSet<T extends Ordered<T>> {
  public abstract boolean isEmpty() ;
  public abstract boolean contains(T element) ;
  public abstract SortedSet<T> add(T element) ;

  public static final <E extends Ordered<E>>
SortedSet<E> empty() {
    return new EmptySet<E>();
  }
}

final class EmptySet<T extends Ordered<T>>
extends SortedSet<T> {
  public boolean isEmpty() {
    return true ;
  }
  public boolean contains(T element) {
    return false ;
  }
  public SortedSet<T> add(T element) {
    return new Node<T>(this,element,this) ;
  }

  public EmptySet() {
  }
}

final class Node<T extends Ordered<T>> extends
SortedSet<T> {

  private final SortedSet<T> left ;
  private final T element ;
  private final SortedSet<T> right ;

  public boolean isEmpty() {
    return false ;
  }
  public Node(SortedSet<T> left, T element,
SortedSet<T> right) {
    this.left = left ;
    this.right = right ;
    this.element = element ;
  }
  public boolean contains(T needle) {
    if (needle.isLessThan(this.element)) {
      return this.left.contains(needle) ;
    } else if (this.element.isLessThan(needle)){
      return this.right.contains(needle) ;
    } else {
      return true ;
    }
  }
  public SortedSet<T> add(T newGuy) {
    if (newGuy.isLessThan(this.element)) {
      return new Node<T>(left.add(newGuy),this.
element,right) ;
    } else if (this.element.isLessThan(newGuy))
{
      return new Node<T>(left,this.
element,right.add(newGuy)) ;
    } else {
      return this ; // Already in set.
    }
  }
}
```

Be warned that the Java library's `Set` interface (optionally) allows imperative addition and removal of elements. A computational rendering of mathematics cannot use these features.

A run-time set might also be backed by an immutable hash table.

Regardless of representation, these set data structures typically need to support operations like enumeration, union, intersection and difference, and relations like membership, equality, and subset.

Whether a balanced tree or a hash map is better for ease of implementation and performance rests on type of the elements in the set and the algorithmic uses-cases for the set operations.

In some cases, it's easy to provide an efficient ordering function. Sometimes, it's easier to provide a hash function and a definition of equality.

Python provides syntactic support for hash-backed sets:

```
>>> { 3 , 2 , 1 } == { 1 , 2 , 3 }
True


>>> {1,2,3} | {3,4,5}
set([1, 2, 3, 4, 5])
```

Racket also provides native sets:

```
> (equal? (set 3 2 1) (set 1 2 3))
#t


> (set-union (set 3 2 1) (set 3 4 5))
(set 1 2 3 4 5)
```

In Haskell, `Data.Set` provides a full-featured implementation of sorted, balanced tree-backed sets.

I'm fond of the following notation for Haskell:

```
import Data.Set


type P = Data.Set.Set
```

so that I can write things like:

```
type Ints = P(Int)
```

which is aesthetically closer to the formal mathematics.

Sets as predicates

If the set X is a subset of Y, but the structure of the set X is outside the descriptive capacity of the type system, then X could be represented as a predicate:

```
class Y {
  public boolean isX() { ... }
}
```

or in Haskell:

```
isX :: Y -> Bool
```

Some advanced programming languages like Agda support dependent types, which allow predicates in the type system itself.

In Racket, rich, expressive contracts take the place of dependent types.

Disjoint union (sums)

A disjoint (or tagged) union of several sets is a new set containing all of the elements of the constituent sets, but with an implicit mark (or tag) added to each element to indicate from which constituent set it came.

The set $A + B$ is the disjoint union of the sets A and B.

In mathematics, that distinguishing mark is almost always kept implicit or inferred from context. (The tag is rarely needed.)

In fact, when that mark is required, it is common to use syntactic sets.

In Java (and other object-oriented languages), sum types are represented through class-based inheritance. The sum forms an abstract base type, and each constituent forms a subtype. For example, the type $A + B + C$ would become:

```
abstract class ApBpC { ... }


class A extends ApBpC { ... }
class B extends ApBpC { ... }
class C extends ApBpC { ... }
```

Haskell supports algebraic data types that closely mimic the sum form. Explicit constructors serve as tags. For example:

```
data ApBpC = ACons A
           | BCons B
           | CCons C
```

The constructors are also used for pattern-matching; for example:

```
whatAmI (ACons _) = "I'm an A."
whatAmI (BCons _) = "I'm a B."
whatAmI (CCons _) = "I'm a C."
```

Of course, in Java, a `whatAmI` method becomes dynamic dispatch:

```
abstract class ApBpC {
  abstract public String whatAmI() ;
}
class A extends ApBpC {
  public String whatAmI() {
    return "I'm an A." ;
  }
}
class B extends ApBpC {
  public String whatAmI() {
    return "I'm a B." ;
  }
}
class C extends ApBpC {
  public String whatAmI() {
    return "I'm a C." ;
  }
}
```

In untyped languages like Racket, where the universal type is already the sum of all types, there is no need for a special embedding.

Languages like Python can exploit class-based inheritance or take the Racket-like approach for representing sum types.

Sequences and vectors

Sequences are a common discrete structure, and their rendering in code is perhaps the most straightforward.

In formal notation, the set of sequences over elements in S is written S^*.

It is usually clear from context whether S^* should contain infinite sequences or only finite ones. (And, in many cases, it doesn't matter.)

For example, if the set A = {`a, b`} is an alphabet, then the set of strings over this alphabet is A^*, which would contain elements like ab and babba.

If the variable used to denote elements of the set S is s, then a sequence in the set S^* is usually a bold-faced $\boldsymbol{s}$ or $\mathbf{s}$. (It is a common convention to use the lower-case version of a set to denote members of that set.)

Given a sequence $\boldsymbol{s}$, its first element is s_1, and its ith element is s_i.

Explicit sequences tend to be wrapped in angle-brackets, so that:

$$\boldsymbol{s} = \langle s_1, s_2, \ldots s_n \rangle$$

Sequences as linked lists

Most frequently, a finite sequence will be encoded as a linked list.

For example, in Java:

```
abstract class Sequence<S> {
  public abstract S getHead() ;
  public abstract Sequence<S> getTail() ;
  public abstract boolean isEmpty() ;

  public static final <T> Sequence<T> cons(T
head, Sequence<T> tail) {
    return new Cons<T>(head,tail) ;
  }

  public static final <T> Sequence<T> nil() {
    return new Nil<T>() ;
  }
}

final class Cons<S> extends Sequence<S> {
  private final S head ;
  private final Sequence<S> tail ;

  public S getHead() {
    return this.head ;
  }

  public Sequence<S> getTail() {
    return this.tail ;
  }

  public boolean isEmpty() {
    return false ;
  }

  public Cons(S head, Sequence<S> tail) {
   this.head = head ;
   this.tail = tail ;
  }
}

final class Nil<S> extends Sequence<S> {
  public S getHead() {
    throw new EmptySequenceException() ;
  }
```

```
  public Sequence<S> getTail() {
    throw new EmptySequenceException() ;
  }

  public boolean isEmpty() {
    return true ;
  }

  public Nil() {  }
}

class EmptySequenceException extends RuntimeEx-
ception {
}

class Test {
  public static void main(String[] args) {
    Sequence<Integer> end = Sequence.nil() ;

    Sequence<Integer> si =
     Sequence.cons(3, end) ;
  }
}
```

Functional languages excel at encoding sequences. Haskell has a list type: `[]`. A function that adds one to every element of a list could be written:

```
add1 :: [Int] -> [Int]
add1 [] = []
add1 (hd:tl) = (hd + 1):(add1 tl)
```

Or, more succinctly using map:

```
add1 :: [Int] -> [Int]
add1 = map (+1)
```

In most Lisps (like Racket), `cons` constructs lists, while `car` and `cdr` destruct:

```
(car (cons 1 (cons 2 '())))  ==  1
(car (cdr (cons 1 (cons 2 '()))))  ==  2
(pair? '())  ==  #f
(pair? (cons 1 '()))  ==  #t
```

In Python, tuples and lists work about equally well as sequences, but tuples might be less error-prone since they're immutable.

Of course, the standard warning about side effects applies: do not use the side-effecting features of Python lists, like popping an element.

Vectors as arrays

When dealing with a set of sequences which all have the same length, the term "vector" is commonly used instead of "sequence."

The set of vectors over the set S of length n is written S^n.

Sometimes vectors are written with a right-arrow ($\rightarrow$) over the unbolded representative variable.

Vectors can be efficiently encoded using arrays, but lists also suffice.

Remember: the array must not be mutated!

If you need to update an index in a vector, it should be copied into a new array first, leaving the original array untouched.

That said, it is often the case that you can prove that when one vector is computed as the update of another vector that the original vector is garbage. In these cases, it is a safe and common optimization to mutate the array.

Infinite sequences as streams

Infinite sequences are not common, but when they arise, they are often encoded as streams.

In Haskell, laziness means that any list can be an infinite list.

It is easy to encode an infinite sequence like the list of all natural numbers:

```
nats = 1:(map (+1) nats)
```

so that `take 5 nats` yields `[1,2,3,4,5]`.

And, even more remarkably, we can filter this list to produce the list of all primes:

```
isPrime n = all (\ m -> n `mod` m /= 0) [2..n-1]

primes = filter isPrime (tail nats)
```

It is actually the case that `take 6 primes` yields `[2,3,5,7,11,13]`.

Racket includes a stream library, allowing the equivalent:

```
(define (inc n) (+ 1 n))
(define nats (stream-cons 1 (stream-map inc nats)))
(define (prime? n)
  (call/ec (λ (return)
    (for ([m (in-range 2 (- n 1))])
      (when (= (modulo n m) 0)
        (return #f)))
    (return #t))))
(define primes (stream-filter prime? (stream-rest
nats)))
```

In an object-oriented setting like Python or Java, streams can be constructed from an interface:

```
interface Stream<A> {
  public A first() ;
  public Stream<A> rest() ;
}
```

The `first()` method should be sure to cache its result, and if the stream is I/O-backed, then the `rest()` method should invoke the `first()` method.

Cartesian products (tuples)

Cartesian products, or tuples, are ordered collections, where the location of the element in the collection determines its type.

Cartesian products map onto records, structs, and objects, with each index into the tuple occupying a field.

For instance, $A \times B$ produces a set of pairs, where the first element is from the set A, and the second is from the set B.

Individual tuples are denoted with parentheses.

For example, (a, b, c) is a member of $A \times B \times C$. In Java, the type $A \times B$ would be a class:

```
class AtimesB {
  public final A a ;
  public final B b ;
  public AtimesB(A a, B b) {
    this.a = a ;
    this.b = b ;
  }
}
```

In Racket, this would be a struct:

```
(define-struct a×b (a b))
```

Python contains native tuple support:

```
>>> x = (1,1,2,3)
>>> x[3]
3
```

But, one might just as easily have defined a class:

```
class MyTuple:
  def __init__(self,first,second,third,fourth):
    self.first = first ;
    self.second = second ;
    self.third = third ;
    self.fourth = fourth ;
```

Haskell provides native tuple support, too:

```
Prelude> let t = (1,2,3)
Prelude> t
(1,2,3)
```

Haskell also allows for record-like data types, such as in the following two definitions:

```
data AB = Pair A B
data AB' = Pair' { a :: A, b :: B }
```

Both definitions introduce constructors:

```
Pair :: A -> B -> AB
Pair' :: A -> B -> AB'
```

The second definition introduces two extractors, one for each field:

```
a :: AB' -> A
b :: AB' -> B
```

Functions (maps)

Mathematical functions transform inputs to outputs.

The set of functions from the set A into the set B is the set $A \to B$.

Under the interpretation of $(\to)$ as an operator on sets, the signature

$$f : X \to Y$$

can be interpreted as the function f is a member of the set $X \to Y$:

$$f \in X \to Y$$

In mathematical notation, there are several extant notations for application:

$$f(x) = f\,x = f.x$$

All of these are the application of the function f to the value x.

In code, functions can translate into procedures and methods, but if they're finite, they can also translate into finite maps backed by hash tables or sorted, balanced tree maps.

Functions as code

Most of the time a mathematical function will map into a procedure in the underlying language.

When a function is supposed to map into executable code, it's usually straightforward to make the mapping using the data structures and algorithms presented elsewhere in this guide.

Functions as maps

In some cases, mathematicians use functions as the formal analog of a hash table or a dictionary. For example:

$$f\,[x \mapsto y]$$

represents a function identical to f except that x maps to y.

Please note that extending a function like this does not (and cannot) change the original function f!

Immutable red-black tree maps are a great data structure for representing these finite functions meant to be extended.

Once again, it is not safe to use the mutable sorted maps and hash tables provided by the Java library, nor the mutable dictionaries provided by Python.

Haskell provides the `Data.Map` library for this purpose, and Racket also offers immutable hash maps.

Sometimes, it is acceptable to hijack the native notion of functions to get them to act like immutable dictionaries. For instance, in Python, we can define extend:

```
def extend (f, x, y):
  return lambda xx: y if xx == x else f(xx)
def empty(x): raise Exception("No such input")
```

so that the following works:

```
g = extend(extend(empty, 3, 4), 5, 6)
print(g(3)) # prints 4
print(g(5)) # prints 6
```

The disadvantage of taking over the internal notion of function like this is that one cannot enumerate the contents of the function, as with a hash or tree-backed formulation.

Immutable maps atop mutable structures

If a language already provides a good native map-like structure (like Python's dictionaries), then it is possible to exploit this structure through shallow copies every time the map is extended:

```
class DictMap:
  def __init__(self, contents):
    self.contents = contents
  def extend(self,x,y):
    contents_ = copy.copy(self.contents)
    contents_[x] = y
    return DictMap(contents_)
  def __call__(self,x):
    return self.contents[x]
```

Relations

Structurally, a relation R is a (possibly infinite) set of subset of some Cartesian product.

The set of all relations over $A \times B$ is $P(A \times B)$.

Computational encodings of relations center on understanding relations in terms of other data structures.

In the special case where a relation relates elements of the same set, e.g. $R \subseteq A \times A$, then R defines a directed graph over nodes in A.

Given a relation R, the notation

$$R(x_1,\ldots,x_n)$$

is equivalent to

$$(x_1,\ldots,x_n) \in R.$$

There are three common ways to encode a relation computationally: as a collection, as a function, and as a predicate.

Relations as collections

Structurally, a relation is a set of tuples, and for finite relations, an encoding as a set of tuples is reasonable.

Relations as functions

Given a relation $R \subseteq A \times B$, the following congruences allow a relation to be represented as a function:

$$P(A \times B) \cong A \to P(B).$$

This functional encoding of a relation is particularly popular for implementing the transition relation of abstract machines, which relates a machine state to all of its possible successors.

Relations as predicates

If one only needs to know whether or not some tuple is within the relation, then it is frequently most efficient to encode the relation as a predicate.

This view is supported by another congruence:

$$P(A \times B) \cong A \times B \to \{\textbf{true},\textbf{false}\}$$

Syntax

Syntactic sets are common within the fields of formal methods and programming languages.

A syntactic definition for the set E uses the following form:

$$E ::= pat_1 \mid \ldots \mid pat_n$$

where each syntactic pattern pat defines a new syntactic form for constructing members of the set E.

A syntactic set is, in essence, a disjoint union with explicit tags.

Viewing syntactic sets as sum types guides translation into code.

Syntactic set examples

For example, we can define a syntax for expression trees:

$$\begin{array}{lll} E & ::= & e + e \\ & \mid & e * e \\ & \mid & n \end{array}$$

We might then define an evaluator eval : $E \to N$ on this set:

$$\begin{array}{ll} eval(e + e) & = eval(e) + eval(e) \\ eval(e * e) & = eval(e) \times eval(e) \\ eval(n) & = n \end{array}$$

In Java (or any object-oriented language), this could become:

```
abstract class Exp {
  abstract public int eval() ;
}

class Sum extends Exp {
  public final Exp left ;
  public final Exp right ;

  public Sum(Exp left, Exp right) {
    this.left = left ;
    this.right = right ;
  }

  public int eval() {
    return left.eval() + right.eval() ;
  }
}
```

```
class Product extends Exp {
  public final Exp left ;
  public final Exp right ;

  public Product(Exp left, Exp right) {
    this.left = left ;
    this.right = right ;
  }

  public int eval() {
    return left.eval() * right.eval() ;
  }
}

class Const extends Exp {
  public int value ;

  public Const(int value) {
    this.value = value ;
  }

  public int eval() {
    return value ;
  }
}
```

To define a sum type with explicit tags, one might use the following form:

$$\begin{array}{lll} Kont & ::= & \mathbf{letk}(\mathrm{v}, \mathrm{e}, \rho, \kappa) \\ & \mid & \mathbf{seqk}(\mathrm{e}, \rho, \kappa) \\ & \mid & \mathbf{setk}(\mathrm{v}, \mathrm{e}, \rho, \kappa) \\ & \mid & \mathbf{halt} \end{array}$$

In Haskell, this structure could be:

```
data Kont = LetK Var Exp Env Kont
          | SeqK Exp Env Kont
          | SetK Var Exp Env Kont
          | Halt
```

In mathematics, the syntactic notation can only be used if the representative variables for each set (e.g., κ for *Kont*, ρ for *Env*) have been clearly established, since in the Haskell notation, these types are required. ■

Matt Might is a professor of Computer Science at the University of Utah. His research interests include programming language design, static analysis and compiler optimization. He blogs at *matt.might.net/articles* and tweets from *@mattmight*.

Reprinted with permission of the original author.
First appeared in *hn.my/mathcode* (matt.might.net)

Hiring Employee #1

By JASON COHEN

It's a big decision to make your first hire, because what you're really deciding is whether you want to keep a lifestyle business or attempt to "cross the chasm" and maybe even get rich.

Assuming you really are in the market for another pair of hands to screw stuff up worse than you already do, the question is how to acquire resumes, how to pare them down, and how to identify someone who is going to work well in your company.

There's already a lot of great advice about hiring at little startups. Before I give you mine, here are some of my favorite articles, in no particular order:

- "Smart, And Gets Things Done" [hn.my/joelhire] by Joel Spolsky. The classic guide to what to do during the interview and how to know whether to "hire" or "not hire."
- "Hazards of Hiring" [hn.my/hazards] by Eric Sink. Great tips, including some specific to hiring developers.
- "Why Startups Should Always Compromise When Hiring" [hn.my/compromise] by Dharmesh Shah. There are many attributes you'd like to see in a hire, but compromise is necessary; here's how to do it.
- "Five Quick Pointers on Startup Hiring" [hn.my/5points] and "Disagreeing with Entrepreneur Magazine" [hn.my/disagree] by Dharmesh Shah. Assorted tips, all important.
- "Date Before Getting Married," Part 1 [hn.my/married1] and Part 2 [hn.my/married2], by Dharmesh Shah. A strong argument in favor of working with a person rather than relying on interviews.

I'm not going to rehash those or attempt a "complete guide to hiring."

But I do have some fresh advice you might not have seen before:

Hire "Startup-minded" People

If a person just left IBM, is she a good fit for your startup?

If she left because she couldn't stand the crushing bureaucracy, the tolerance of incompetence, and the lack of any visibility into what customers actually wanted, then she sounds like a person ready for a startup.

Or therapy.

On the other hand, if during the interview she asks how often you do performance reviews, that means she doesn't understand the startup culture. If she says "I thrive in environments with clear requirements, written expectations, and defined processes," run away as fast as your little legs can carry you.

Startups are chaotic. Rules change, and there is no "job description." It's better to make a strong decision that turns out wrong, and admit it, than to plan ahead or wait for instructions. Potential earnings (e.g. stock, performance bonuses) are preferred to guaranteed earnings (e.g. salary, benefits).

You already live by this Code of Turmoil because you're the entrepreneur; you have no choice. But normal people do have a choice, and most abhor chaos. Big companies don't behave this way, and most people are accustomed to working for big companies.

You have to hire someone comfy with the bedlam of startup life.

Write a Crazy Job Description

You're not just hiring any old programmer or salesman, you're hiring employee #1. This person helps set the culture of the company. This person has to mesh with your personality 100%. You're going to be putting in long hours together. If they don't get your jokes, it's not going to work.

So, why wait until the interview to see whether your personalities mesh? Put it right in the job description.

Be funny, reflect your personality, and reflect the uniqueness of your company. See the jobs page at WP Engine [wpengine.com/careers] for a bunch of examples — everything from detailing our culture ("Being transparent about our strengths and weaknesses wins us sales") to attitude on writing awesome code ("You think using a profiler is fun, like a treasure hunt") to treating customers ("Whether or not you sleep at night is directly proportional to whether you've made someone thrilled or pissed off that day").

You should see the results in the cover letters. If after a job posting like that the person is still sending the generic bullshit cover letter, you know they're not for you. If they respond in kind, good sign.

And anyway, one day you actually might need them to change those pellets, and then you've got it in writing!

Do Not Use a Recruiter

On young startups using recruiters, Bryan Menell sums it up nicely:

> *"If you find yourself wanting to hire a recruiter, hit yourself in the head with a frying pan until the feeling goes away."*

You need to hire an absolute superstar, and recruiters are not in the business of helping you find superstars.

In fact, their incentives are exactly opposite yours. Here's why.

Recruiters are like real estate salesmen: they make money when you hire someone. They make the same amount of money whether it takes you four days or four months to find that someone. So every day that passes, every additional resume you request, every additional interview you set up, the recruiter is making less and less money per hour.

In fact, there's a floor that the recruiter can't go below, so the more time you take to find the right person, the more they'll push you to settle for someone you've already rejected.

The exception is a recruiter who works by the hour rather than for a hiring bounty. These are hard to find, but they do exist. I've had luck only in this case.

Resumes Are (Mostly) Useless

Think about your own resume. Is there anything on there that qualifies you to run your own company? Not just "experience" generically, but really relevant knowledge? I'll bet there's very little. But it doesn't matter, right?

Right; so it doesn't matter with your first few employees either.

Resumes are useful only as talking points. That is, when you have a candidate on the phone, you can use the resume to ask about previous experience, test their knowledge of technologies they claim to have, etc. Resumes are conversation-starters, but they imply nothing about whether the person is right for you.

One particularly useful trick with resumes is to dig deep on a detail. Pick the weirdest technology in the list, or pick on one bullet point they listed two jobs ago that seems a little odd to you. Then go deep. Don't let them say "It's been a while" — if they can't talk about it, how can they claim it's experience they're bringing along?

> **“In a small startup there’s no layer separating employees from customers. You can’t have your company represented by someone who can’t be trusted with a customer.”**

Writing Skills Are Required

I don’t care if this person is going to spend 60 hours a week writing inscrutable code that only a Ruby interpreter could love. I don’t care if the job description is “sit in that corner and work multi-variate differential equations.” Everyone has to be able to communicate clearly.

In a modern startup everyone will be writing blog entries, Twittering, Facebooking, and God only knows what the hell other new Goddamn technology is coming next. But whatever it is, you can bet it will require good communication skills.

In a small startup there’s no layer separating employees from customers. Everyone talks to everyone. You can’t have your company represented by someone who can’t be trusted with a customer. In fact, everyone needs to be able to not just talk to customers, but even sell to them. Remember, tech support is sales!

In a small startup everyone has to understand one other’s nuances. There’s enough crap you have to figure out without also having to decipher an email. There’s enough about your business you don’t understand without translating garbage sentence fragments in a README file.

Therefore, some part of the interview process has to include free-form writing. In fact, there’s a particularly useful time for that….

Screen Candidates With Mini-Essay Questions

When you post a job listing — especially on large-scale sites like Monster or Craig’s List — expect a torrent of resumes. It’s not unusual to get 100 in a day. You need a time-efficient system for winnowing them down to a small handful worthy of an interview.

Screening resumes is not an option, because resumes are useless. Besides, you don’t have time to read hundreds of resumes.

Instead, prepare an email template that asks the applicant to write a few paragraphs on a few topics. For example:

> *Thanks for sending us your resume. The next step in our hiring process is for you to write a few paragraphs on each of the following topics. Please reply to this email address with your response:*
>
> 1. *Why do you want to work at [company]?*
> 2. *Describe a situation in your work-life where you failed.*
> 3. *Describe a time when you accomplished something you thought was impossible. (Can be work-related or personal)*
>
> *Thanks for your interest in [company], and I hope to hear from you soon.*

Here’s what happens: First, most people never respond. Good riddance! Second, you’ll get lazy-ass responses like “I want to work at your company because I saw you are hiring” and ludicrous answers like “I have never failed at anything.”

Resist the temptation to reply with, “You just did.” That’s what assholes do.

Maybe 10% of the respondents will actually answer the questions, and you’ll know in two minutes whether this person can communicate and, yes, even whether they seem fun, intelligent, or interesting.

One exception to this rule: If the cover letter is truly wonderful, that’s a rare, great sign, and you can probably skip right to the phone interview.

Always Be Hiring

The rule of thumb is that it takes 3-6 months to hire a really good person. Why so long?

- Good people are rare, so it takes a while to dig them up. Like truffles. Or weeds.
- Good people won't change jobs more often than once a year — probably more like every 3 – 4 years, especially if their employer appreciates their abilities and compensates them accordingly. So you have to find this person in their "once every three years" window.
- Good people gets lots of good job offers (yes, even in this economy). So when you do find one, and you give them the writing test, the phone interview,,the in-person interview, discuss compensation, and then provide a formal written offer… there's a good chance they just accepted an awesome offer somewhere else. (This happened to me all the time at Smart Bear. It's happening now at WP Engine.)

This means if you start hiring when you really need someone, that's too late. You'll be "in need" for months.

This means you need to be hiring constantly.

So how do you "hire constantly" without being drowned in resumes and interviews? The answer comes from another attribute of good people:

Good people choose where they want to work, not vice versa. They hear about a cool company, and when they're interested in new work, they call you. Your company has to be a place good people will seek, not where you have to go fishing. How do you manage that, especially when you're small? Ideas:

- Develop your blog/Twitter so you have a steady stream of eyeballs from people who like you.
- Attend local meet-ups and user groups. Meet the woman who runs the group — she knows everyone worth knowing.
- Sponsor a meet-up at your office. Don't have an office? Co-sponsor with someone who does, like another company or a co-working place.
- Ask your friends for resumes of people they didn't hire but who they liked. That is, people who are good but just weren't a fit for that company.
- Try to get your "Jobs" page to rank well in local-only search. So e.g. "java programmer job in Austin TX," not something impossible like "java programmer."
- Take everyone you know to lunch periodically and ask if they know of a candidate. Yes, you can ask them by email, but often being in-person brings out more information. Or maybe one of them will be interested himself. (That's happened to me a few times.)

Don't Be Trapped by What You Think Hiring "Should" Be

You're hiring a friend, a trusted partner, someone you'll be spending 10 hours a day with for the foreseeable future.

You're not hiring a Systems Engineer III for IBM or a Senior Regional Sales Manager for Dell. The "rules" of HR don't apply to you (except the law).

Think of it more like getting married than hiring an underling.

Going with your gut is not wrong. ■

Jason Cohen is the founder of WP Engine [wpengine.com] — Heroku for WordPress, after exitting from three previous companies. He blogs at *blog.asmartbear.com*

Reprinted with permission of the original author. First appeared in *hn.my/hire1* (asmartbear.com)

Edward Tufte's

Slopegraphs

By CHARLIE PARK

Back in 2004, Edward Tufte defined and developed the concept of a "sparkline." Odds are good that if you're reading this, you're familiar with them and how popular they've become.

What's interesting is that over twenty years before sparklines came on the scene, Tufte developed a different type of data visualization that didn't fare nearly as well. To date, in fact, I've only been able to find three examples of it, and even they aren't completely in line with his vision.

It's curious that it hasn't become more popular; the chart type is quite elegant, aligns with all of Tufte's best practices for data visualization, and was created by the master of information design. Why haven't these charts (christened "slopegraphs" by Tufte about a month ago) taken off the way sparklines did?

We're going to look at slopegraphs: what they are, how they're made, why they haven't seen a massive uptake so far, and why I think they're about to become much more popular in the near future.

Photo: Christmas #32, *flickr.com/photos/pagedooley/5274477590*

The Table-Graphic

In his 1983 book The Visual Display of Quantitative Information [hn.my/visual], Tufte displayed a new type of data graphic.

As Tufte notes in his book, this type of chart is useful for seeing:

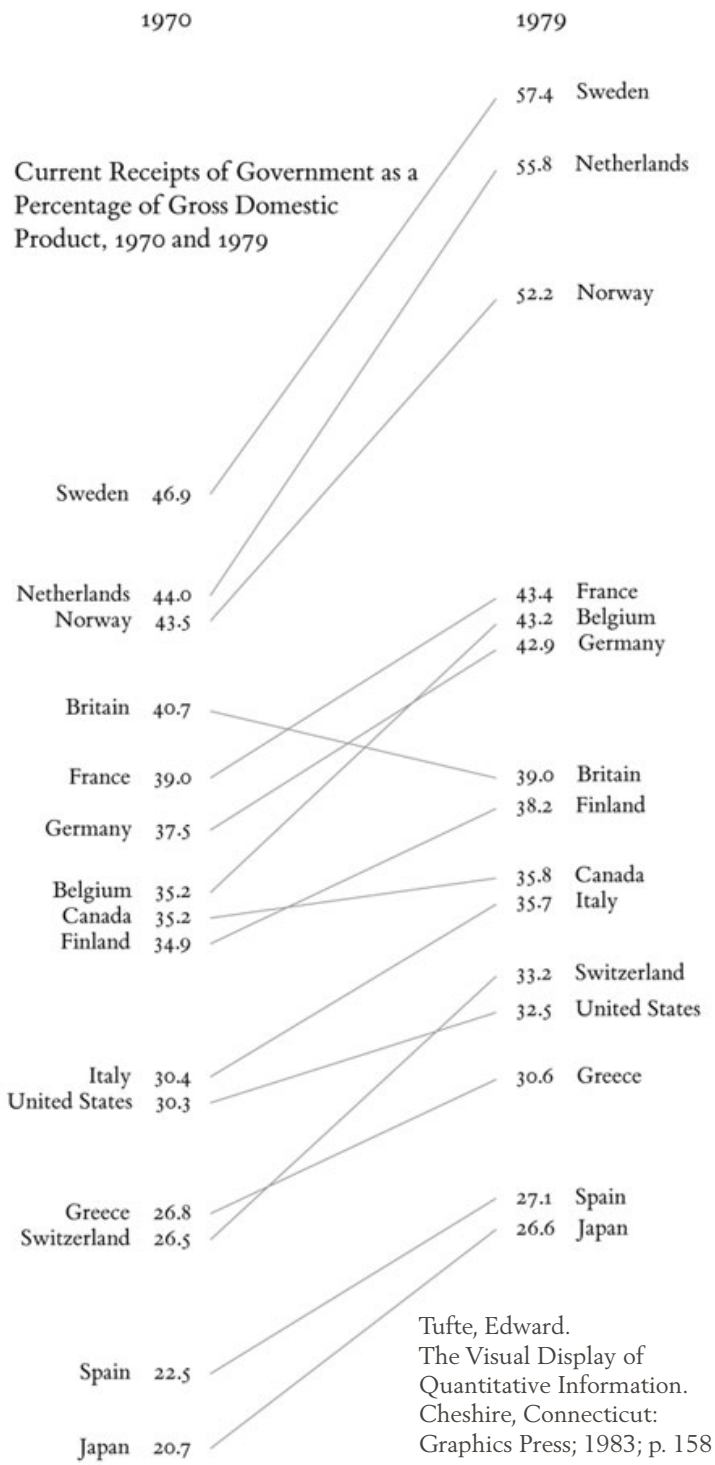
- the hierarchy of the countries in both 1970 and 1979 [the order of the countries]
- the specific numbers associated with each country in each of those years [the data value next to their names]
- how each country's numbers changed over time [each country's slope]
- how each country's rate of change compares to the other countries' rates of change [the slopes compared with one another]
- any notable deviations in the general trend (notice Britain in the above example) [aberrant slopes]

This chart does this in a remarkably minimalist way. There's absolutely zero non-data ink.

So, anyway, Professor Tufte made this new kind of graph. Unlike sparklines, though, it didn't really get picked up. Anywhere.

My theory on this lack of response is three-fold:

1. It didn't have a name. (He just referenced it as a "table-graphic" at the time.)
2. It was a totally new concept. (Where sparklines are easily understood as "an axis-less line chart, scaled down (and kind of cute)," this "table-graphic" is something new.)
3. It's a little good deal more complicated to draw. (More on that at the end.)



Tufte, Edward. The Visual Display of Quantitative Information. Cheshire, Connecticut: Graphics Press; 1983; p. 158

A Super-Close Zoom-In on a Line Chart

The best way I've found to describe these table-graphics is this: it's like a super-close zoom-in on a line chart, with a little extra labeling.

Imagine you have a line chart, showing the change in European countries' population over time. Each country has a line, zigzagging from January (on the left) to December (on the right). Each country has twelve points across the chart. The lines zigzag up and down across the chart.

Now, let's say you zoomed in to just the June-July segment of the chart, and you labeled the left and right sides of each country's June-July lines (with the country's name, and the specific number at each data point).

That's it. Fundamentally, that's all a table-graphic is.

Hierarchical Table-Graphics in the Wild

Where sparklines found their way into products at Google (Google Charts and Google Finance) and Microsoft, and even saw some action from a pre-jQuery John Resig (jspark.js [hn.my/jspark]), this table-graphic thing saw essentially zero uptake.

At present, Googling for "tufte table-graphic" yields a whopping eighty-three results, most of which have nothing to do with this technique.

Actually, since Tufte's 1983 book, I've found three non-Tuftian examples (total). And even they don't really do what Tufte laid out with his initial idea.

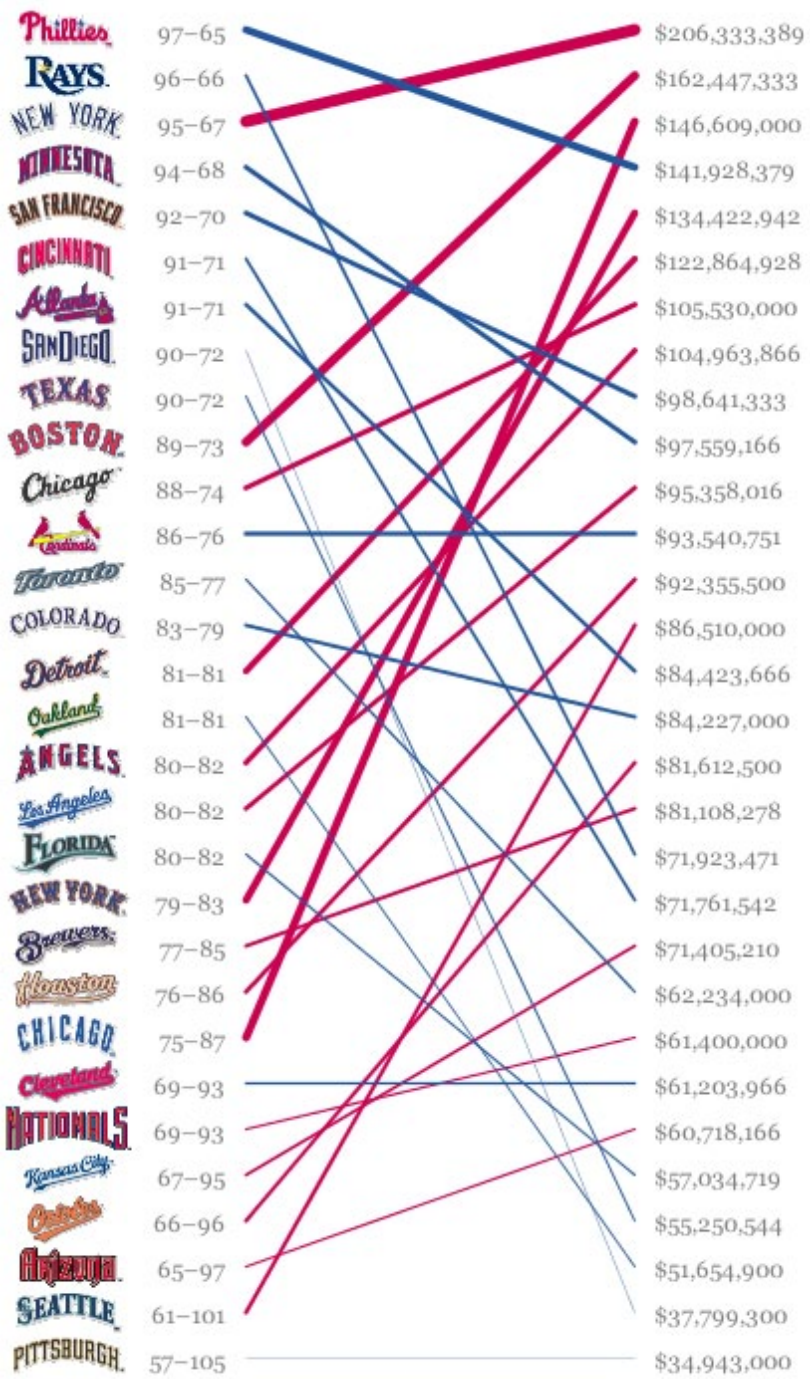
Let's look at each of them.

Ben Fry's Baseball Chart

The first we'll look at came from Processing developer/data visualization designer Ben Fry, who developed a chart showing baseball team performance vs. total team spending:

A version of this graphic was included in his 2008 book Visualizing Data, but I believe he shared it online before then.

Anyway, you can see each major-league baseball team on the left, with their win/loss ratio on the left and their annual budget on the right. Between them is a sloped line showing how their ordering in each column compares. Lines angled up (red) suggest a team that is spending more than their win ratio suggests they should be, where blue lines suggest the team's getting a good value for their dollars. The steeper the blue line, the more wins-per-dollar.



[benfry.com/salaryper/]

There are two key distinctions between Tufte's chart and Fry's chart.

First: Fry's baseball chart is really just comparing order, not scale. The top-most item on the left is laid out with the same vertical position as the top-most item on the right, and so on down the list.

Second: Fry's is comparing two different variables: win ratio and team budget. Tufte's looks at a single variable, over time. (To be fair, Fry's does show the change over time, but only in a dynamic, online version, where the orders change over time as the season progresses. The static image above doesn't concern itself with change-over-time.)

If you want to get technical, Fry's chart is essentially a "forced-rank parallel coordinates plot" with just two metrics.

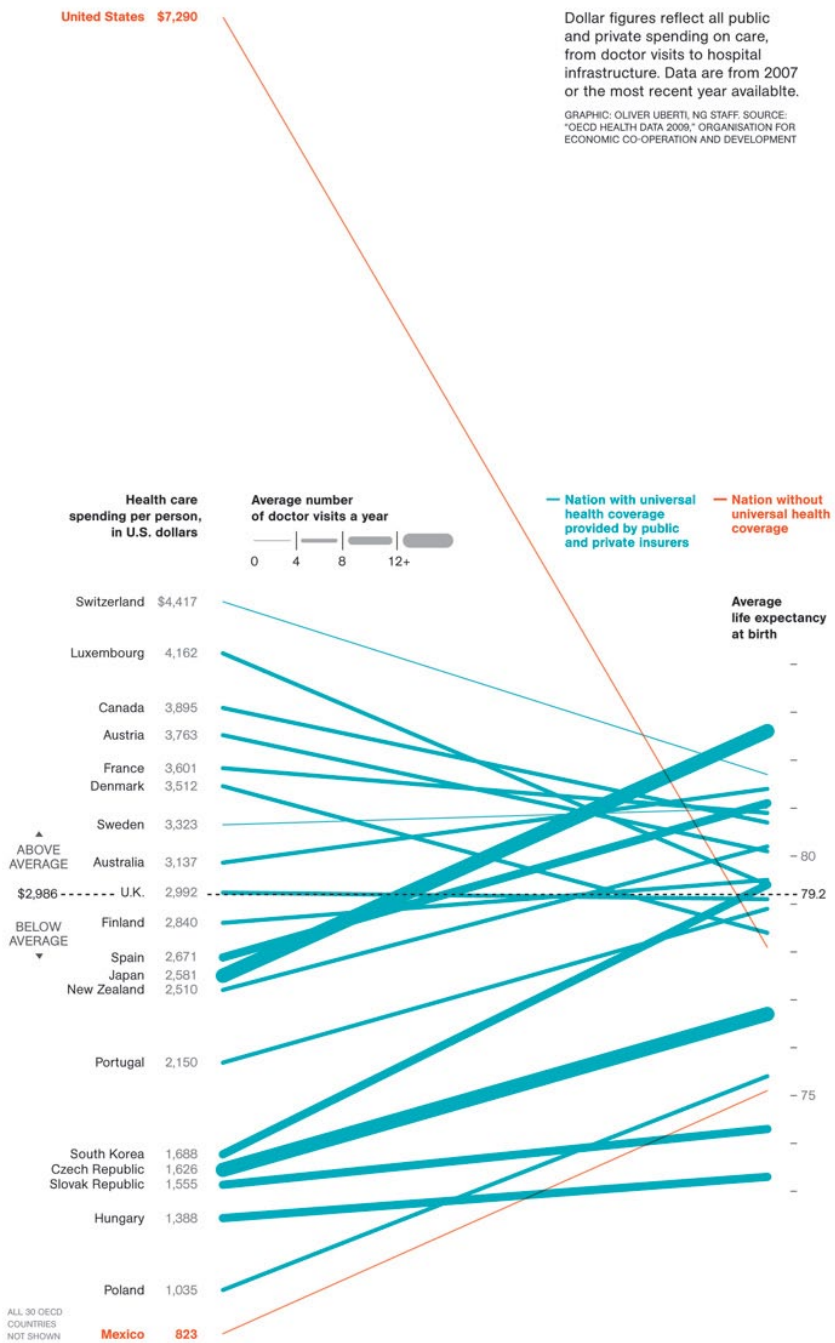
Another difference I should note: This type of forced-rank chart doesn't have any obvious allowance for ties. That is, if two items on the chart have the same datum value (as is the case in eleven of the thirty teams above), the designer (or the algorithm, if the process is automated) has to choose one item to place above the other. (For example, see the Reds and the Braves, at positions #6 and #7 on the left of the chart.) In Fry's case, he uses the team with the lower salary as the "winner" of the tie. But this isn't obvious to the reader.

In Visualizing Data, Fry touches on the "forcing a rank" question (p. 118), noting that at the end of the day, he wants a ranked list, so a scatterplot using the X and Y axes is less effective of a technique (as the main point with a scatterplot is simply to display a correlation, not to order the items). I'm not convinced, but I am glad he was intentional about it. I also suspect that,because the list is generated algorithmically, it was easier to do it and avoid label collisions this way.

Nevertheless, I do think it's a good visualization.

The National Geographic Magazine Life-Expectancy Chart

In 2009, Oliver Uberti at National Geographic Magazine released a chart showing the average life expectancy at birth of citizens of different countries, comparing that with what each nation spends on health care per person:



[blogs.ngm.com/blog_central/2009/12/the-cost-of-care.html]

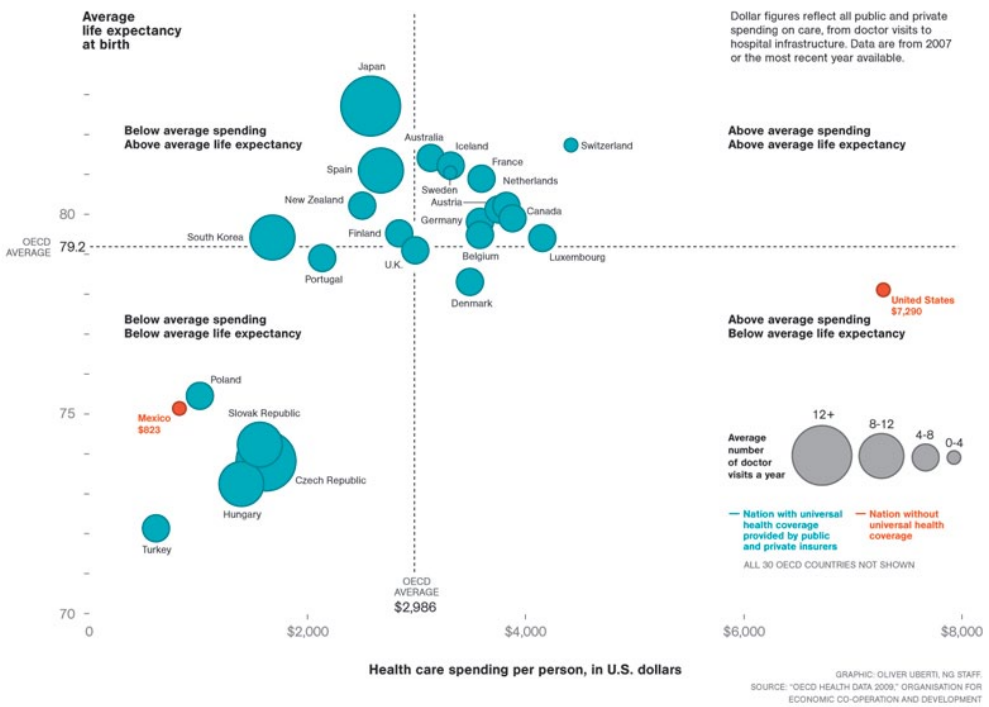
Like Fry's chart, Uberti's chart uses two different variables. Unlike Fry's chart, Uberti's does use different scales. While that resolves the issue I noted about having to force-rank identical data points, it introduces a new issue: dual-scaled axes.

By selecting the two scales used, the designer of the graph — whether intentionally or not — is introducing meaning where there might not actually be any.

For example, should the right-side data points have been spread out so that the highest and lowest points were as high and low as the Switzerland and Mexico labels (the highest and lowest figures, apart from the US) on the left? Should the scale been adjusted so that the Switzerland and/or Mexico lines ran horizontally? Each of those options would have affected the layout of the chart. I'm not saying that Uberti should have done that — just that a designer needs to tread very carefully when using two different scales on the same axis.

A few bloggers criticized the NatGeo chart, noting that, like the Fry chart above, it was an Inselberg-style parallel-coordinates plot, and that a better option would be a scatter plot.

In a great response on the NatGeo blog [hn.my/natgeo], Uberti then re-drew the data in a scatter plot:



Uberti also gave some good reasons for drawing the graph the way he did originally, with his first point being that "many people have difficulty reading scatter plots. When we produce graphics for our magazine, we consider a wide audience, many of whose members are not versed in visualization techniques. For most people, it's considerably easier to understand an upward or downward line than relative spatial positioning."

I agree with him on that. Scatterplots reveal more data, and they reveal the relationships better (and Uberti's scatterplot is really good, apart from a few quibbles I have about his legend placement). But scatterplots can be tricky to parse, especially for laymen.

Note, for example, that in the scatter plot, it's hard at first to see the cluster of bubbles in the bottom-left corner of the chart, and the eye's initial "read" of the chart is that a best-fit line would run along that top-left-to-bottom-right string of bubbles from Japan to Luxembourg. In reality, though, that line would be absolutely wrong, and the best-fit would run from the bottom-left to the upper-right.

Also, the entire point of the chart is to show the US's deviant spending pattern, but in the scatter plot, the eye's activity centers around that same cluster of bubbles, and the US's bubble on the far right is lost.

The "Above average spending / Below average life expectancy" labels on the quadrants are really helpful, but, again, it reinforces Uberti's point, that scatter plots are tricky to read. Should those labels really be necessary? Without them, would someone be able to glance at the scatter chart and "get it"?

For quick scanning, the original chart really does showcase the extraordinary amount the US spends on healthcare relative to other countries. And that's the benefit of these table-graphics: slopes are easy to read.

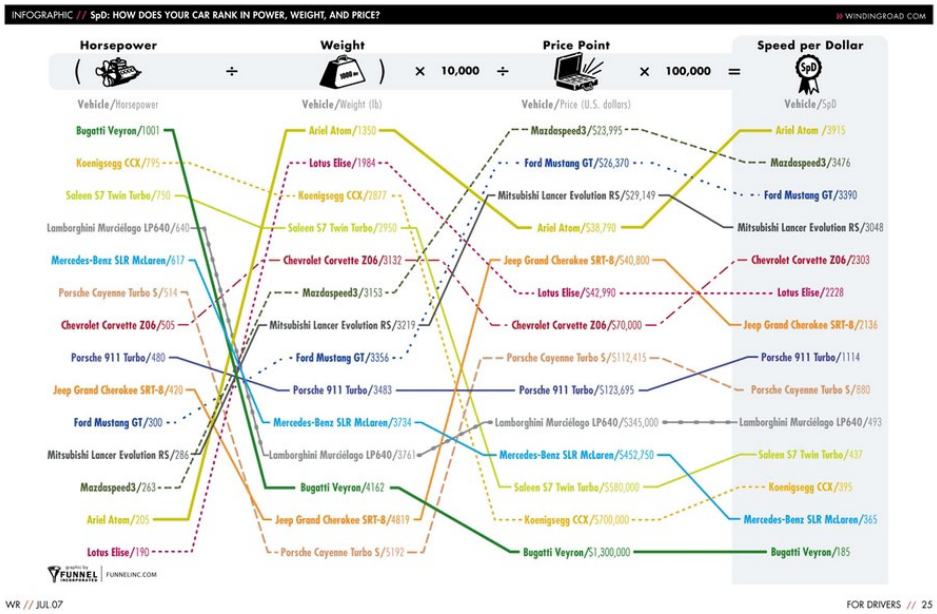
Speed Per Dollar

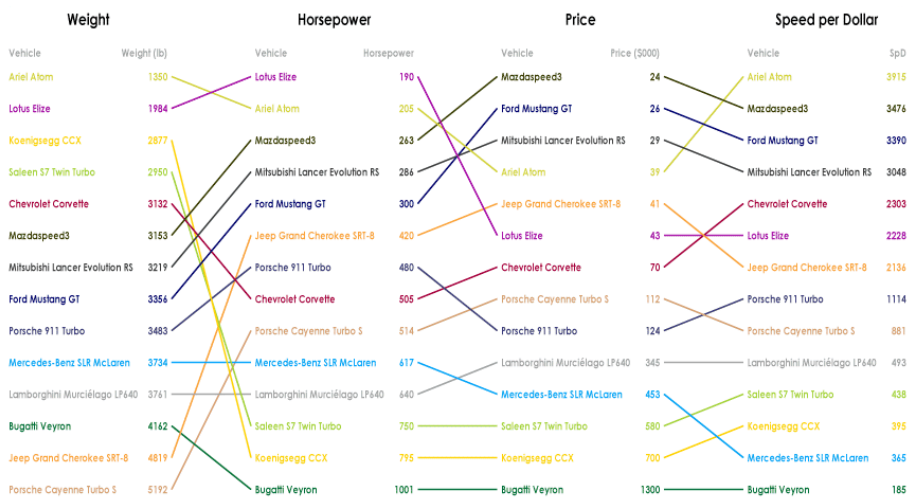
Back in July of 2007 (I know: we're going back in time a bit, but this chart diverges even more from Tufte's than the others, and I wanted to build up to it), a designer at online driving magazine WindingRoad.com developed the "Speed per Dollar" index (on the right).

Again, what we have is, essentially, an Inselberg-style parallel-coordinates plot, with a Fry-style forced-rank. In this case, though, each step of the progression leads us through the math, to the conclusion at the right-side of the chart: dollar-for-dollar, your best bet is the Ariel Atom.

Anyway, this chart uses slopes to carry meaning, hence its inclusion here, but I think it's different enough from the table-chart Tufte developed in 1983 that it isn't quite in the same family.

Dave Nash, a "kindly contributor" at Tufte's forum then refined the chart, making aspects of it clearer and more Tuftian (on the right).





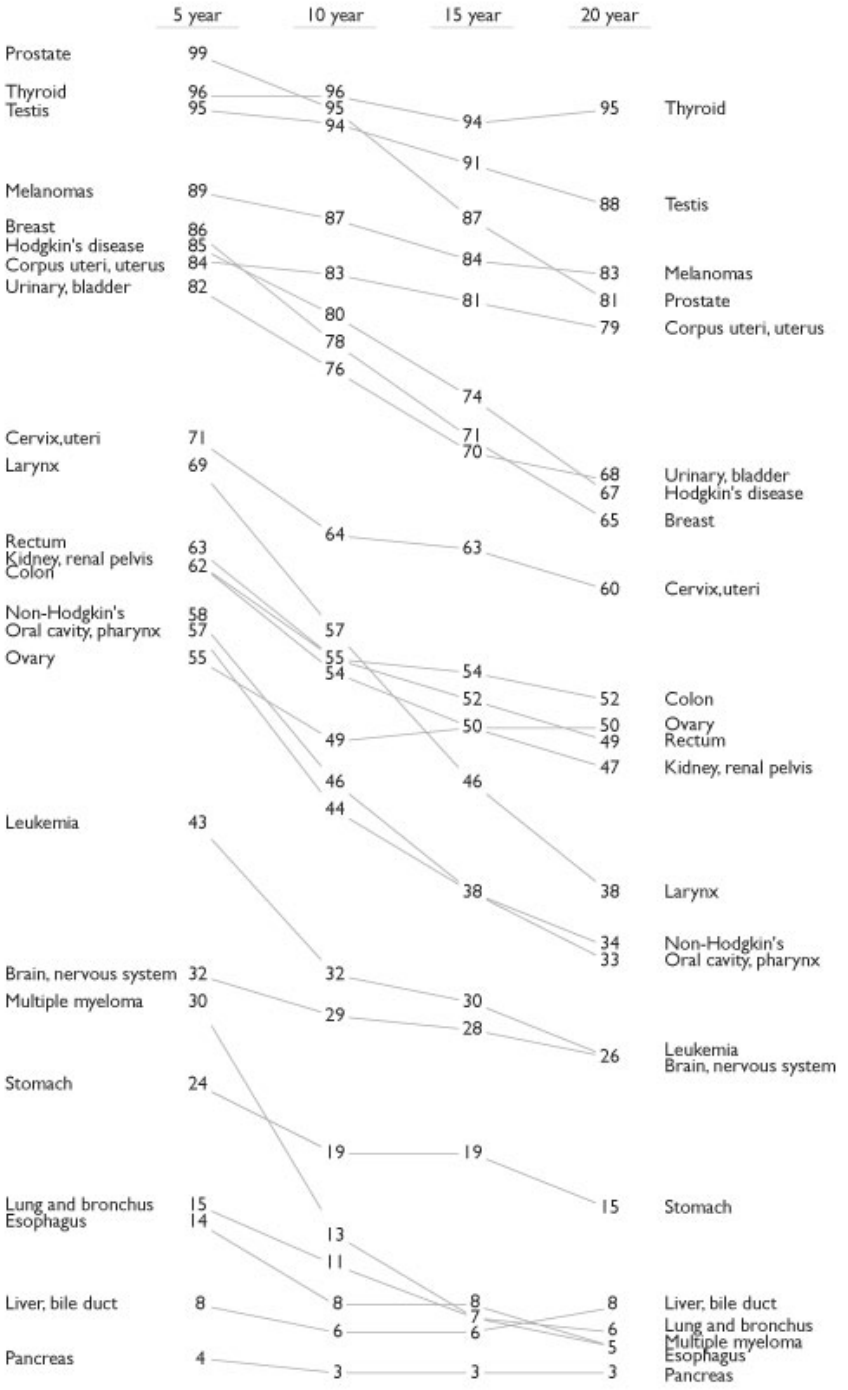
(I like how the original included the math at the top of the chart, showing how the SPD value was derived, and I like how it highlights the final column, drawing the eye to the conclusions, but I do think Nash's shows the data better.)

Cancer Survival Rates

We'll close with the last example of these table-charts I've found.

This one's from Tufte himself. It shows cancer survival rates over 5-, 10-, 15-, and 20-year periods.

Actually, the chart below is a refinement of a Tufte original (2002), done (again) by Kindly Contributor Dave Nash (2003, 2006).



[edwardtufte.com/bboard/q-and-a-fetch-msg?msg_id=0000Jr]

Owing it to being a creation of the man himself, this is most in-line with the table-chart I showed at the very top, from 1983. We can clearly see each item's standings on the chart, from one quinquennium to the next. In fact, this rendition of the data is a good illustration of my earlier simplification, that these table-charts are, essentially, minimalist versions of line charts with intra-line labels.

Tufte Names His Creation

Although it's possible that Tufte has used this term in his workshops, the first occasion I can find of the "table-chart" having an actual name is this post from Tufte's forums [hn.my/tufteforum] on June 1st, 2011. The name he gives the table-chart: **Slopegraphs**.

I suspect that we'll see more slopegraphs in the wild, simply because people will now have something they can use to refer to the table-chart besides "that slopey thing Tufte had in Visual Design."

But there's still a technical problem: How do you make these damn things?

Making Slopegraphs

At the moment, both of the canonical slopegraphs were made by hand, in Adobe Illustrator. A few people have made initial efforts at software that aids in creating slopegraphs. It's hard, though. If the labels are too close together, they collide, making the chart less legible. A well-done piece of software, then, is going to include collision-detection and account for overlapping labels in some regard.

Here are a few software tools that are currently being developed:

- Dr. David Ruau has developed a working version of slopegraphs in R [hn.my/slopegraphr].
- Alex Kerin, slopegraphs in Tableau [hn.my/tableau].
- Coy Yonce, slopegraphs in Crystal Reports [hn.my/slopecr] and slopegraphs in Crystal Reports Enterprise [hn.my/slopecre].
- I started developing a slopegraphs in Javascript/Canvas version [hn.my/slopejs], but probably won't continue it, and Ill try to use the Google Charts' API if I try again. The "jaggies" on the lines were too rough for me.

In each case, if you use the chart-making software to generate a slopegraph, attribute the software creator.

With this many people working on software implementations of slopegraphs, I expect to see a large uptick in slopegraphs in the next few months and years. But when should people use slopegraphs?

When to Use Slopegraphs

In Tufte's June 1st post, he sums up the use of slopegraphs well: "Slopegraphs compare changes over time for a list of nouns located on an ordinal or interval scale."

Basically: Any time you'd use a line chart to show a progression of univariate data among multiple actors over time, you might have a good candidate for a slopegraph. There might be other occasions where it would work as well. Note that strictly by Tufte's June 1st definition, none of the examples I gave (Baseball, Life Expectancy, Speed-per-Dollar) count as slopegraphs.

But some situations clearly would benefit from using a slopegraph, and I think Tufte's definition is a good one until more examples come along and expand it or confirm it.

An example of a good slopegraph candidate: in my personal finance webapp PearBudget, we've relied far more on tables than on charts. (In fact, the only chart we include is a "sparkbar" under each category's name, showing the amount of money available in the current month.) We've avoided charts in general (and pie charts in particular, unlike every other personal finance webapp), but I'm considering adding a visual means of comparing spending across years — how did my spending on different categories this June compare with my spending on those categories in June of 2010? Did they all go up? Did any go down? Which ones changed the most? This would be a great situation in which to use a slopegraph.

Slopegraph Best Practices

Because slopegraphs don't have a lot of uses in place, best practices will have to emerge over time. For now, though:

- Be clear. First to yourself, then to your reader, whether your numbers are displaying the items in order or whether they're on an actual scale.
- If the data points or labels are bunching up, expand the vertical scale as necessary.
- Left-align the names of the items on both the left-hand and right-hand axes to make vertical scanning of the items' names easier.
- Include both the names of the items and their values on both the left-hand and right-hand axes.
- Use a thin, light gray line to connect the data. A too-heavy line is unnecessary and will make the chart harder to read.
- But when a chart features multiple slope intersections (like the baseball or speed-per-dollar charts above), judicious use of color can avoid what Ben Fry describes as the "pile of sticks" phenomenon (Visualizing Data, 121).
- A table with more statistical detail might be a good complement to use alongside the slopegraph. As Tufte notes: "The data table and the slopegraph are colleagues in explanation not competitors. One display can serve some but not all functions."
- Defer to current best practices outlined by Tufte, Stephen Few, and others, including maximizing data-to-ink ratios, minimizing chartjunk, and so on. ■

Charlie Park runs the online personal finance app *pearbudget.com*, and is about to launch *monotask.com*, described as "ADD meds for your computer." He lives in Virginia with his wife and three daughters.

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

How to Read Haskell Like Python

By EDWARD Z. YANG

HAVE YOU EVER been in a situation where you need to quickly understand what a piece of code in some unfamiliar language does? If the language looks a lot like what you're comfortable with, you can usually guess what large amounts of the code do, even if you may not be familiar with how all the language features work.

For Haskell, this is a little more difficult, since Haskell syntax looks very different from traditional languages. But there's no really deep difference here; you just have to squint at it right. Here is a fast, mostly incorrect, and hopefully useful guide for interpreting Haskell code like a Pythonista. By the end, you should be able to interpret this fragment of Haskell (some code elided with ...):

```
runCommand env cmd state = ...
retrieveState = ...
saveState state = ...

main :: IO ()
main = do
    args <- getArgs
    let (actions, nonOptions, errors) = getOpt Permute options args
    opts <- foldl (>>=) (return startOptions) actions
    when (null nonOptions) $ printHelp >> throw NotEnoughArguments
    command <- fromError $ parseCommand nonOptions
    currentTerm <- getCurrentTerm
    let env = Environment
            { envCurrentTerm = currentTerm
            , envOpts = opts
            }
    saveState =<< runCommand env command =<< retrieveState
```

- **Types.** Ignore everything you see after `::` (similarly, you can ignore `type`, `class`, `instance` and `newtype`. Some people claim that types help them understand code. If you're a complete beginner, things like `Int` and `String` will probably help, and things like `LayoutClass` and `MonadError` won't. Don't worry too much about it.)
- **Arguments.** `f a b c` translates into `f(a, b, c)`. Haskell code omits parentheses and commas. One consequence of this is we sometimes need parentheses for arguments: `f a (b1 + b2) c` translates into `f(a, b1 + b2, c)`.
- **Dollar sign.** Since complex statements like `a + b` are pretty common and Haskellers don't really like parentheses, the dollar sign is used to avoid parentheses: `f $ a + b` is equivalent to the Haskell code `f (a + b)` and translates into `f(a + b)`. You can think of it as a big opening left parenthesis that automatically closes at the end of the line (no need to write `))))))` anymore!). In particular, if you stack them up, each one creates a deeper nesting: `f $ g x $ h y $ a + b` is equivalent to `f (g x (h y (a + b)))` and translates into `f(g(x,h(y,a + b))` (though some consider this bad practice).

In some code, you may see a variant of `$: <$>` (with angled brackets). You can treat `<$>` the same way as you treat `$`. (You might also see `<*>`; pretend that it's a comma, so `f <$> a <*> b` translates to `f(a, b)`. There's not really an equivalent for regular `$`)

- **Backticks.** `x `f` y` translates into `f(x,y)`. The thing in the backticks is a function, usually binary, and the things to the left and right are the arguments.
- **Equals sign.** Two possible meanings. If it's at the beginning of a code block, it just means you're defining a function:

```
doThisThing a b c = ...
  ==>
def doThisThing(a, b, c):
  ...
```

Or if you see it near a `let` keyword, it's acting like an assignment operator:

```
let a = b + c in ...
  ==>
a = b + c
...
```

- **Left arrow.** Also acts like an assignment operator:

```
a <- createEntry x
  ==>
a = createEntry(x)
```

Why don't we use an equals sign? Shenanigans. (More precisely, `createEntry x` has side effects. More accurately, it means that the expression is monadic. But that's just shenanigans. Ignore it for now.)

- **Right arrow.** It's complicated. We'll get back to them later.
- **Do keyword.** Line noise. You can ignore it. (It does give some information, namely that there are side effects below, but you never see this distinction in Python.)
- **Return.** Line-noise. Also ignore. (You'll never see it used for control flow.)
- **Dot.** `f . g $ a + b` translates to `f(g(a + b))`. Actually, in a Python program you'd probably have been more likely to see:

```
x = g(a + b)
y = f(x)
```

But Haskell programmers are allergic to extra variables.

- **Bind and fish operators.** You might see things like `=<<`, `>>=`, `<=<` and `>=>`. These are basically just more ways of getting rid of intermediate variables:

```
doSomething >>= doSomethin-
gElse >>= finishItUp
  ==>
x = doSomething()
y = doSomethingElse(x)
finishItUp(y)
```

Sometimes a Haskell programmer decides that it's prettier if you do it in the other direction, especially if the variable is getting assigned somewhere:

```
z <- finishItUp =<< doSome-
thingElse =<< doSomething
  ==>
x = doSomething()
y = doSomethingElse(x)
z = finishItUp(y)
```

The most important thing to do is to reverse engineer what's actually happening by looking at the definitions of `doSomething`, `doSomethingElse` and `finishItUp`: it will give you a clue what's "flowing" across the fish operator. If you do that, you can read `<=<` and `>=>` the same way (they actually do function composition, like the dot operator). Read `>>` like a semicolon (e.g. no assignment involved):

```
doSomething >> doSomethin-
gElse
  ==>
doSomething()
doSomethingElse()
```

- **Partial application.** Sometimes, Haskell programmers will call a function, but they won't pass enough arguments. Never fear; they've probably arranged for the rest of the arguments to be given to the function somewhere else. Ignore it, or look for functions which take anonymous functions as arguments. Some of the usual culprits include `map`, `fold` (and variants), `filter`, the composition operator `.`, the fish operators (`=<<`, etc). This happens a lot to the numeric operators: `(+3)` translates into lambda `x: x + 3`.
- **Control operators.** Use your instincts on these: they do what you think they do! (Even if you think they shouldn't act that way.) So if you see: `when (x == y) $ doSomething x`, it reads like "When x equals y, call doSomething with x as an argument."

  Ignore the fact that you couldn't actually translate that into `when(x == y, doSomething(x))` (Since, that would result in doSomething always being called.) In fact, `when(x == y, lambda: doSomething x)` is more accurate, but it might be more comfortable to just pretend that when is also a language construct.

  `if` and `case` are built-in keywords. They work the way you'd expect them to.

- **Right arrows (for real!)** Right arrows have nothing to do with left arrows. Think of them as colons: they're always nearby the `case` keyword and the backslash symbol, the latter of which is lambda: `\x -> x` translates into `lambda x: x`.

  Pattern matching using `case` is a pretty nice feature, but a bit hard to explain here. Probably the easiest approximation is an `if..elif..else` chain with some variable binding:

```
case moose of
  Foo x y z -> x + y * z
  Bar z -> z * 3
  ==>
if isinstance(moose, Foo):
  x = moose.x # the variable binding!
  y = moose.y
  z = moose.z
  return x + y * z
elif isinstance(moose, Bar):
  z = moose.z
  return z * 3
else:
  raise Exception("Pattern match failure!")
```

- **Bracketing.** You can tell something is a bracketing function if it starts with `with`. They work like contexts do in Python:

```
withFile "foo.txt" ReadMode $ \h -> do
  ...
  ==>
with open("foo.txt", "r") as h:
  ...
```

  (You may recall the backslash from earlier. Yes, that's a lambda. Yes, `withFile` is a function. Yes, you can define your own.)

- **Exceptions.** `throw`, `catch`, `catches`, `throwIO`, `finally`, `handle` and all the other functions that look like this work essentially the way you expect them to. They may look a little funny, however, because none of these are keywords: they're all functions, and follow all those rules. So, for example:

```
trySomething x `catch` \(e :: IOException) ->
handleError e
  ===
catch (trySomething x) (\(e :: IOException) ->
handleError e)
  ==>
try:
  trySomething(x)
except IOError as e:
  handleError(e)
```

- **Maybe.** If you see `Nothing`, it can be thought of as `None`. So `isNothing x` tests if `x is None`. What's the opposite of it? `Just`. For example, `isJust x` tests if `x is not None`.

  You might see a lot of line noise associated with keeping `Just` and `None` in order. Here's one of the most common ones:

```
maybe someDefault (\x -> ...) mx
  ==>
if mx is None:
  x = someDefault
else:
  x = mx
...
```

  Here's one specific variant, for when a null is an error condition:

```
maybe (error "bad value!") (\x -> ...) x
  ==>
if x is None:
  raise Exception("bad value!")
```

- **Records.** The work they way you'd expect them too, although Haskell lets you create fields that have no names:

```
data NoNames = NoNames Int Int
data WithNames = WithNames {
  firstField :: Int,
  secondField :: Int
}
```

  So `NoNames` would probably be represented as a tuple `(1, 2)` in Python, and `WithNames` a class:

```
class WithNames:
  def __init__(self, firstField, secondField):
    self.firstField = firstField
    self.secondField = secondField
```

  Then creation is pretty simple `NoNames 2 3` translates into `(2, 3)`, and `WithNames 2 3` or `WithNames { firstField = 2, secondField = 3 }` translates into `WithNames(2,3)`.

  Accessors are a little more different. The most important thing to remember is Haskellers put their accessors before the variable, whereas you might be most familiar with them being after. So `field x` translates to `x.field`. How do you spell `x.field = 2`? Well, you can't really do that. You can copy one with modifications though:

```
return $ x { field = 2 }
  ==>
y = copy(x)
y.field = 2
return y
```

  Or you can make one from scratch if you replace `x` with the name of the data structure (it starts with a capital letter). Why do we only let you copy data structures? This is because Haskell is a pure language; but don't let that worry you too much. It's just another one of Haskell's quirks.

- **List comprehensions.** They originally came from the Miranda-Haskell lineage! There are just more symbols.

```
[ x * y | x <- xs, y <- ys, y > 2 ]
  ==>
[ x * y for x in xs for y in ys if y > 2 ]
```

  It also turns out Haskellers often prefer list comprehensions written in multi-line form (perhaps they find it easier to read). They look something like:

```
do
  x <- xs
  y <- ys
  guard (y > 2)
  return (x * y)
```

  So if you see a left arrow and it doesn't really look like it's doing side effects, maybe it's a list comprehension.

- **More symbols.** Lists work the way you would expect them to in Python; `[1, 2, 3]` is in fact a list of three elements. A colon, like `x:xs` means construct a list with `x` at the front and `xs` at the back (`cons`, for you Lisp fans.) `++` is list concatenation. `!!` means indexing. Backslash means lambda. If you see a symbol you don't understand, try looking for it on Hoogle [haskell.org/hoogle](yes, it works on symbols!).

- **More line noise.** The following functions are probably line noise, and can probably be ignored. `liftIO`, `lift`, `runX` (e.g. `runState`), `unX` (e.g. `unConstructor`), `fromJust`, `fmap`, `const`, `evaluate`, an exclamation mark before an argument (`f !x`), `seq`, a hash sign (e.g. `I# x`).

- **Bringing it all together.** Let's return to the original code fragment:

```
runCommand env cmd state = ...
retrieveState = ...
saveState state = ...

main :: IO ()
main = do
    args <- getArgs
    let (actions, nonOptions, errors) = getOpt Permute
options args
    opts <- foldl (>>=) (return startOptions) actions
    when (null nonOptions) $ printHelp >> throw NotEn-
oughArguments
    command <- fromError $ parseCommand nonOptions
    currentTerm <- getCurrentTerm
    let env = Environment
            { envCurrentTerm = currentTerm
            , envOpts = opts
            }
    saveState =<< runCommand env command =<<
retrieveState
```

With some guessing, we can pop out this translation:

```
def runCommand(env, cmd, state):
   ...
def retrieveState():
   ...
def saveState(state):
   ...

def main():
  args = getArgs()
  (actions, nonOptions, errors) = getOpt(Permute(),
options, args)
  opts = **mumble**
  if nonOptions is None:
     printHelp()
     raise NotEnoughArguments
  command = parseCommand(nonOptions)

  currentTerm = getCurrentTerm()
  env = Environment(envCurrentTerm=currentTerm,
envOpts=opts)

  state = retrieveState()
  result = runCommand(env, command, state)
  saveState(result)
```

This is not bad, for a very superficial understanding of Haskell syntax (there's only one obviously untranslatable bit, which requires knowing what a fold is. Not all Haskell code is folds; I'll repeat, don't worry about it too much!)

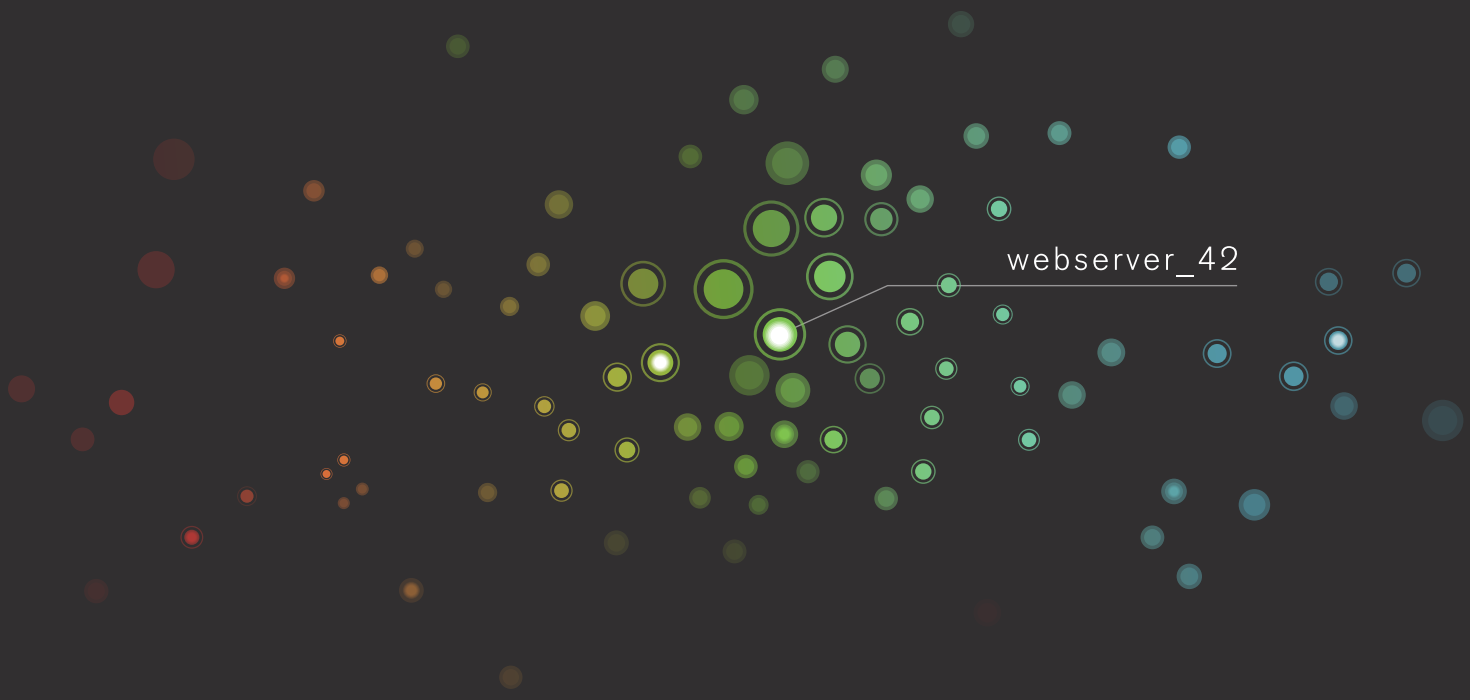
Most of the things I have called "line noise" actually have very deep reasons behind them, and if you're curious about the actual reasons behind these distinctions, I recommend learning how to write Haskell. But if you're just reading Haskell, I think these rules should be more than adequate. ■

Edward Z. Yang is an undergraduate currently studying computer science at MIT/University of Cambridge. He is interested in topics related to functional programming, and plays the oboe.

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

These are your servers

These are your servers on Cloudkick



Any questions?

cloudkick.com
415.779.5425

support for 8 clouds + dedicated hardware

cloudkick

the best way to manage the cloud

Fast, Easy, Realtime Metrics Using Redis Bitmaps

By CHANDRA PATNI

At Spool, we calculate our key metrics in real time. Traditionally, metrics are performed by a batch job (running hourly, daily, etc.). Redis-backed bitmaps allow us to perform such calculations in realtime and are extremely space efficient. In a simulation of 128 million users, a typical metric such as "daily unique users" takes less than 50 ms on a MacBook Pro and only takes 16 MB of memory. Spool doesn't have 128 million users yet, but it's nice to know our approach will scale. We thought we'd share how we do it, in case other startups find our approach useful.

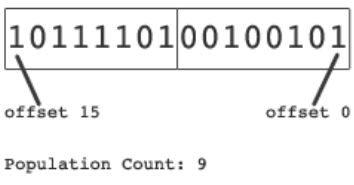
Crash Course on Bitmap and Redis Bitmaps

Bitmap (aka Bitset)

A Bitmap or bitset is an array of zeros and ones. A bit in a bitset can be set to either `0` or `1`, and each position in the array is referred to as an offset. Operations such as logical AND, OR, XOR, etc., and other bitwise operations are fair game for Bitmaps.

Population Count

The population count of a Bitmap is the number of bits set to `1`. There are efficient algorithms for calculating population count. For instance, the population count of a 90% filled bitset containing 1 billion bits took 21.1 ms on a MacBook Pro. There is even a hardware instruction in SSE4 for the population count of an integer.

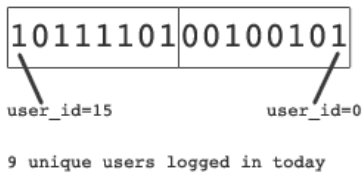


Bitmaps in Redis

Redis allows binary keys and binary values. Bitmaps are nothing but binary values. The setbit(`key`, `offset`, `value`) operation, which takes `O(1)` time, sets the value of a bit to `0` or `1` at the specified offset for a given key.

A simple example: Daily Active Users

To count unique users that logged in today, we set up a bitmap where each user is identified by an offset value. When a user visits a page or performs an action, which warrants it to be counted, set the bit to `1` at the offset representing user id. The key for the bitmap is a function of the name of the action user performed and the timestamp.



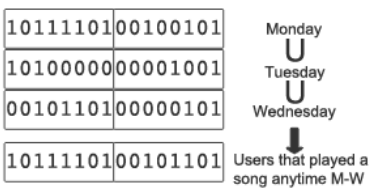
In this simple example, every time a user logs in we perform a `redis.setbit(daily_active_users, user_id, 1)`. This flips the appropriate offset in the daily_active_users bitmap to 1. This is an O(1) operation. Doing a population count on this results in 9 unique users that logged in today. The key is `daily_active_users` and the value is `1011110100100101`.

Of course, since the daily active users will change every day, we need a way to create a new bitmap every day. We do this by simply appending the date to the bitmap key. For example, if we want to calculate the daily unique users who have played at least 1 song in a music app for a given day, we can set the key

name to be `play:yyyy-mm-dd`. If we want to calculate the number of unique users playing a song each hour, we can name the key name `play:yyyy-mm-dd-hh`. For the rest of the discussion, we will stick with daily unique users that played a song. To collect daily metrics, we will simple set the user's bit to 1 in the `play:yyyy-mm-dd` key whenever a user plays a song. This is an O(1) operation.

```
redis.setbit(play:yyyy-mm-dd, user_id, 1)
```

The unique users that played a song today are the population count of the bitmap stored as the value for the `play:yyyy-mm-dd` key. To calculate weekly or monthly metrics, we can simply compute the union of all the daily Bitmaps over the week or the month, and then calculate the population count of the resulting bitmap.



You can also extract more complex metrics very easily. For example, the premium account holders who played a song in November would be:

```
(play:2011-11-01
∪ play:2011-11-02
∪...∪play:2011-11-30) ∩
premium:2011-11
```

Performance comparison using 128 million users

The table below shows a comparison of daily unique action calculations calculated over 1 day, 7 days, and 30 days for 128 million users. The 7 and 30 metrics are calculated by combining daily bitmaps.

Period	Time (ms)
Daily	50.2
Weekly	392.0
Monthly	1624.8

Optimizations

In the above example, we can optimize the weekly and monthly computations by caching the calculated daily, weekly, and monthly counts in Redis.

This is a very flexible approach. An added bonus of caching is that it allows fast cohort analysis, such as weekly unique users who are also mobile users — the intersection of a mobile users bitmap with a weekly active users bitmap. Or, if we want to compute rolling unique users over the last n days, having cached daily unique counts makes this easy; simply grab the previous `n-1` days from your cache and union it with the real time daily count, which only takes 50 ms.

Sample Code

A Java code snippet below computes unique users for a given user action and date.

```
import redis.clients.jedis.Jedis;
import java.util.BitSet;
...
  Jedis redis = new Jedis("localhost");
...
  public int uniqueCount(String action,
String date) {
    String key = action + ":" + date;
    BitSet users = BitSet.valueOf(redis.
get(key.getBytes()));
    return users.cardinality();
  }
```

The code snippet below computes the unique users for a given user action and a list of dates.

```
import redis.clients.jedis.Jedis;
import java.util.BitSet;
...
  Jedis redis = new Jedis("localhost");
...
  public int uniqueCount(String action,
String... dates) {
    BitSet all = new BitSet();
    for (String date : dates) {
      String key = action + ":" + date;
      BitSet users = BitSet.
valueOf(redis.get(key.getBytes()));
      all.or(users);
    }
    return all.cardinality();
  }
```

Chandra Patni is a member of the Spool geek squad. These days he hacks on Node, CoffeeScript, Redis and Ruby. He is @cpatni on Twitter and @rubyorchard on GitHub.

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

Asynchronous UIs

the Future of Web User Interfaces

By ALEX MACCAW

It's an interesting time to be working on the frontend now. We have new technologies such as HTML5, CSS3, Canvas and WebGL, all of which greatly increase the possibilities for web application development. The world is our oyster!

However, there's also another trend I've noticed. Web developers are still stuck in the request/response mindset. I call it the "click and wait" approach — where every UI interaction results in a delay before another interaction can be performed. That's the process they've used their entire careers, so it's no wonder most developers are blinkered to the alternatives.

Speed matters — a lot. Or to be precise, *perceived* speed matters a lot. Speed is a critical and often neglected part of UI design, but it can make a huge difference to user experience, engagement, and revenue.

- Amazon: 100 ms of extra load time caused a 1% drop in sales (source: Greg Linden, Amazon).
- Google: 500 ms of extra load time caused 20% fewer searches (source: Marissa Mayer, Google).
- Yahoo!: 400 ms of extra load time caused a 5–9% increase in the number of people who clicked "back" before the page even loaded (source: Nicole Sullivan, Yahoo!).

Yet, despite all this evidence, developers still insist on using the request/response model. Even the introduction of Ajax hasn't improved the scene much, replacing blank loading states with spinners. There's no technical reason why we're still in this state of affairs, it's purely conceptual.

A good example of the problem is Gmail's "sending" notification; how is this useful to people? What's the point of blocking? 99% of the time the email will be sent just fine.

As developers, we should optimize for the most likely scenario. Behavior like this reminds me of Window's balloon notifications, which were awesome for telling you about something you just did:

The Solution

I've been working on this problem, specifically with a MVC JavaScript framework called Spine [spinejs.com], and implementing what I've dubbed *asynchronous user interfaces*, or AUIs. The key to this is that interfaces should be completely non-blocking. Interactions should be resolved instantly; there should be no loading messages or spinners. Requests to the server should be decoupled from the interface.

The key thing to remember is that users don't care about Ajax. They don't give a damn if a request to the server is still pending. They don't want loading messages. Users would just like to use your application without any interruptions.

The Result

AUIs result in a significantly better user experience, more akin to what people are used to on the desktop than the web. Here's an example of an AUI Spine application with a Rails backend [spine-rails3.herokuapp.com].

Notice that any action you take, such as updating a page, is completely asynchronous and instant. Ajax REST calls are sent off to Rails in the background after the UI has updated. It's a much better user experience.

Compare it to the static version of the same application, which blocks and navigates to a new page on every interaction. The AUI experience is a big improvement that will get even more noticeable in larger (and slower) applications.

Have a browse around the source [hn.my/spinerails] to see what's going on, especially the main controller.

Not a Silver Bullet

It's worth mentioning here that I don't think this approach is a silver bullet for all web applications, and it won't be appropriate for all use cases. One example that springs to mind is credit-card transactions, something you'll always want to be synchronous and blocking. However, I do believe that AUIs are applicable the vast majority of the time.

The other point I'd like to make is that not all feedback is bad. Unobtrusive feedback that's actually useful to your users is completely fine, like a spinner indicating when files have synced, or network connections have finished. The key thing is that feedback is useful and doesn't block further interaction.

The Implementation

So how do you achieve these AUIs? There are a number of key principles:

- Move state and view rendering to the client side
- Intelligently preload data
- Asynchronous server communication

Now, these concepts turn the existing server-driven model on its head. It's often not possible to convert a conventional web application into a client side app; you need to set out from the get-go with these concepts in mind as they involve a significantly different architecture.

Moving state to the client side is a huge subject and beyond the scope of this article. For more information on that, you might want to read my book, JavaScript Web Applications [hn.my/jsapp]. Here I want to focus on a specific part of AUIs: asynchronous server communication, or in other words, server interaction that's decoupled from the user interface.

The idea is that you update the client before you send an Ajax request to the server. For example, say a user updated a page name in a CMS. With an asynchronous UI, the name change would be immediately reflected in the application, without any loading or pending messages. The UI is available for further interaction instantly. The Ajax request specifying the name change would then be sent off separately in the background. At no point does the application depend on the Ajax request for further interaction.

For example, let's take a Spine Model called Page. Say we update it in a controller, changing its name:

```
page = Page.find(1)
page.name = "Hello World"
page.save()
```

As soon as you call save(), Spine will perform the following actions:

1. Run validation callbacks and persist the changes to memory
2. Fire the change event and update the user interface
3. Send an Ajax PUT to the server indicating the change

Notice that the Ajax request to the server has been sent after the UI has been updated; in other words, what we've got here is an asynchronous interface.

Synchronizing State

Now this is all very well in principle, but I'm sure you're already thinking of scenarios where this breaks down. Since I've been working with these types of applications for a while, I can hopefully address some of these concerns. Feel free to skip the next few sections if you're not interested in the finer details.

Validation

What if server validation fails? The client thinks the action has already succeeded, so they'll be pretty surprised if subsequently told that the validation had failed.

There's a pretty simple solution to this: client-side validation. Replicate server-side validation on the client side, performing the same checks. You'll always ultimately need server-side validation, since you can't trust clients. But, by also validating on the client, you can be confident a request will be successful. Server-side validation should only fail if there's a flaw in your client-side validation.

That said, not all validation is possible on the client-side, especially validating the uniqueness of an attribute (an operation which requires DB access). There's no easy solution to this, but there is a discussion covering various options in Spine's documentation.

Network Failures and Server Errors

What happens if the user closes their browser before a request has completed? This is fairly simple to resolve: just listen to the window.onbeforeunload event, check to see if Ajax requests are still pending, and, if appropriate, notify the user. Spine's Ajax documentation contains a discussion about this.

```
window.onbeforeunload = ->
  if Spine.Ajax.pending
    '''Data is still being sent
to the server; you may lose
unsaved changes if you close
the page.'''
```

Alternatively, if the server returns an unsuccessful response code, say a 500, or if the network request fails, we can catch that in a global error handler and notify the user. Again, this is an exceptional event, so it's not worth investing too much developer time into. We can just log the event, inform the user, and perhaps refresh the page to re-sync state.

ID Generation

IDs are useful to refer to client-side records, and are used extensively throughout JavaScript frameworks like Backbone [hn.my/backbone] and Spine. However, this throws up a bit of a dilemma: where are the IDs generated, with the server or the client?

Generating IDs on the server has the advantage that IDs are guaranteed to be unique, but generating them on the client side has the advantage that they're available instantly. How do we resolve this situation?

Well, a solution that Backbone uses is generating an internal cid (or client id). You can use this cid temporarily before the server responds with the real identifier. Backbone has a separate record retrieval API, depending on whether you're using a cid, or a real id.

```
Users.getByCid(internalID)
Users.get(serverID)
```

I'm not such a fan of that solution, so I've taken a different tack with Spine. Spine generates pseudo GUIDs internally when creating records (unless you specify an ID yourself). It'll use that ID to identify records from then on. However, if the response from an Ajax create request to the server returns a new ID, Spine will switch to using the server specified ID. Both the new and old ID will still work, and the API to find records is still the same.

Synchronous Requests

The last issue is with Ajax requests that get sent out in parallel. If a user creates a record, and then immediately updates the same record, two Ajax requests will be sent out at the same time, a `POST` and a `PUT`. However, if the server processes the "update" request before the "create" one, it'll freak out. It has no idea which record needs updating, as the record hasn't been created yet.

The solution to this is to pipeline Ajax requests, transmitting them serially. Spine does this by default, queuing up `POST`, `PUT` and `DELETE` Ajax requests so they're sent one at a time. The next request is sent only after the previous one has returned successfully.

Next Steps

So that's a pretty thorough introduction into asynchronous interfaces, and even if you glossed over the finer details, I hope you've been left with the impression that AUIs are a huge improvement over the status quo, and a valid option when building web applications. ■

Alex MacCaw is a JavaScript/Ruby developer, and works at Twitter with the frontend Revenue Team. He also enjoys traveling and writing, and his book JavaScript Web Applications was published by O'Reilly this year.

Reprinted with permission of the original author. First appeared in *hn.my/aui* (alexmaccaw.co.uk)

The Coding Zone

By PAUL STAMATIOU

I'VE LEARNED THERE are three things that set me up for a productive programming session.

Good Music

An endless supply of new beats works wonders. This is the absolute most important thing for me. If I have to context switch every three minutes to find a better song to play, not much is going to get done. Sometimes I'll loop through a Deadmau5 album on Spotify, or listen to a set like Trance Around the World [trancearoundtheworld.com]. While I really enjoy leaks and mashups on Hype Machine [hypem.com/stammy], it is so hit or miss that I end up having to change the track often.

Getting in the coding zone starts by isolating myself from the rest of the world with my headphones. That's also a sign to Akshay, who works a few feet in front of me, that I'm in get-shit-done mode but have Campfire open if he needs anything.

No Chance of Interruption

I must have a seemingly endless block of time at my disposal. If I have a meeting in one hour, that severely limits how much of a zone I can get into. My most productive work tends to happen at odd hours where there is no possible way that I will get a text about going out for lunch, an IM from Olark, or a bunch of emails filling my inbox.

For example, it's early on Sunday morning, my cofounder is sleeping (I've slumped into a nocturnal phase....We're in a no-meetings-until-we-ship-some-new-stuff mode), and I will probably be up until 7 am in a blissful coding rage. Everything is perfect right now.

Organization

I'm never far away from our Trello board [trello.com], my own personal Trello "scratch" board, my trusty Pilot Hi-Tec C Cavalier 0.3 mm pen and browser sketch pad [hn.my/uistencils] on my desk. Anything that crosses my mind worth doing now goes on my sketch pad and anything worth doing later goes on our Trello. I used to hate Trello because I thought it was fugly, but the simplicity has grown on me.

For some reason I can't seem to shake the "to do.txt" file on my desktop. Nowadays it has evolved into more of a scratch pad as well — random snippets of code or copy that I don't need this moment but don't feel like having to browse through github to find later should I need it again. ■

Honorable mentions: A depth charge [hn.my/depth] or cappuccino on my desk, being motivated about what I'm actually building (that's the easy part for me and why I have only ever worked on my start-ups), and having a clean workspace, both in my physical living area and on my computer's desktop and Picplum Dropbox folder.

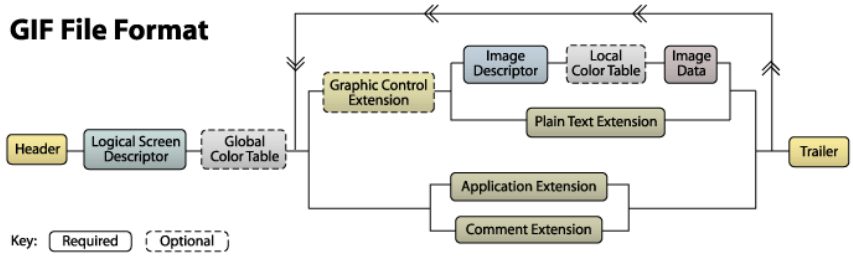
Paul Stamatiou is the co-founder of Picplum [picplum.com], a startup making it easy to automatically send photo prints to loved ones.

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

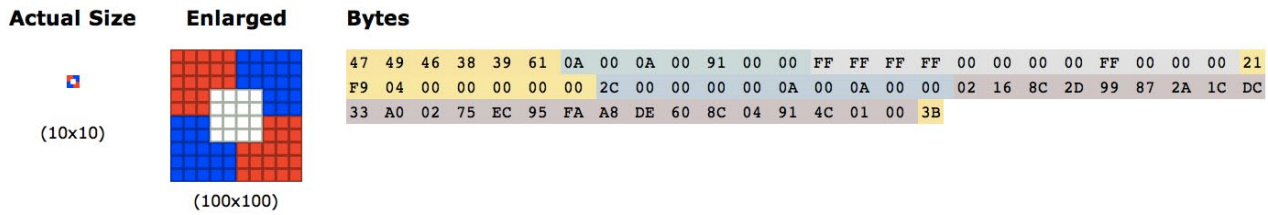
What's in a GIF — Bit by Byte

By MATTHEW FLICKINGER

We will start off by walking though the different parts of a GIF file. (The information on this page is primarily drawn from the W3C GIF89a specification [hn.my/gift89a].) A GIF file is made up of a bunch of different "blocks" of data. The following diagram shows all of the different types of blocks and where they belong in the file. The file starts at the left and works its way right. At each branch you may go one way or the other. The large middle section can be repeated as many times as needed. (Technically, it may also be omitted completely, but I can't imagine what good a GIF file with no image data would be.)



I'll show you what these blocks looks like by walking through a sample GIF file. You can see the sample file and its corresponding bytes on top of next page.



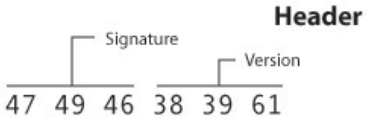
Note that not all blocks are represented in this sample file. I will provide samples of missing blocks where appropriate. The different types of blocks include: header, logical screen descriptor, global color table, graphics control extension, image descriptor, local color table, image data, plain text extension, application extension, comment extension, and trailer. Let's get started with the first block!

Header Block

```
47 49 46 38 39 61
```

(from Sample File)

All GIF files must start with a header block. The header takes up the first six bytes of the file. These bytes should all correspond to ASCII character codes [ascii.cl]. We actually have two pieces of information here. The first three bytes are called the signature. These should always be "GIF" (e.g., 47="G", 49="I", 46="F"). The next three specify the version of the specification that was used to encode the image. We'll only be working with "89a" (e.g., 38="8", 39="9", 61="a"). The only other recognized version string is "87a" but I doubt most people will run into those anymore.



Logical Screen Descriptor

```
0A 00 0A 00 91 00 00
```

(from Sample File)

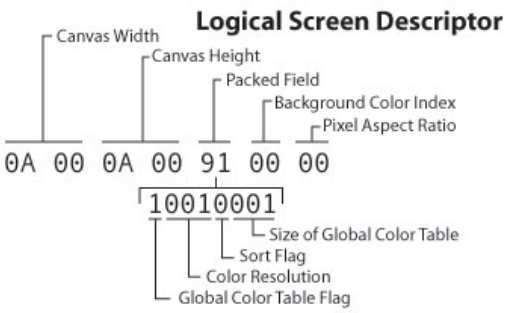
The logical screen descriptor always immediately follows the header. This block tells the decoder how much room this image will take up. It is exactly seven bytes long. It starts with the canvas width. This value can be found in the first two bytes. It's saved in a format the spec simply calls "unsigned." Basically we're looking at a 16-bit, nonnegative integer (0-65,535). As with all the other multi-byte values in the GIF format, the least significant byte is stored first (little-endian format). This means where we would read `0A 00` from the byte stream, we would normally write it as `000A`, which is the same as 10. Thus the width of our sample image is 10 pixels. As a further example, 255 would be stored as `FF 00`, but 256 would be `00 01`. As you might expect, the canvas height follows. Again, in this sample we can see this value is `0A 00`, which is 10.

Next we have a packed byte. That means that this byte actually has multiple values stored in its bits. In this case, the byte 91 can be represented as the binary number `10010001`. (The built-in Windows calculator is actually very useful when converting numbers into hexadecimal and binary formats. Be sure it's in "scientific" or "programmer" mode, depending on the version of Windows you have.) The first and most-significant bit is the global color table flag. If it's 0, then there is none. If it's 1, then a global color table will follow. In our sample image, we can see that we will have a global color table (as will usually be the case). The next three bits represent the color resolution. The spec says this value "is the number of bits per primary color available to the original image, minus 1" and "...represents the size of the entire palette from which the colors in the graphic were selected." Because I don't much about what this one does, I'll point you to a more knowledgeable article on bit and color depth [hn.my/bitcolor]. For now 1 seems to work. Note that 001 represents 2 bits/pixel; 111 would represent 8 bits/pixel. The next single bit is the sort flag. If the value is 1, then the colors in the global color table are sorted in order of "decreasing importance," which typically means "decreasing frequency" in the image. This can help the image decoder but is not required. Our value has been left at 0. The last three bits are the size of global color table. Well, that's a lie; it's not the actual size of the table. If this value is N, then the actual table size is 2^(N+1). From our sample file, we get the three bits 001, which is the binary version of 1. Our actual table size would be

2^(1+1) = 2^2 = 4. (We've mentioned the global color table several times with this byte, we will be talking about what it is in the next section.)

The next byte gives us the background color index. This byte is only meaningful if the global color table flag is 1. It represents which color in the global color table (by specifying its index) should be used for pixels whose value is not specified in the image data. If, by some chance, there is no global color table, this byte should be 0.

The last byte of the logical screen descriptor is the pixel aspect ratio. I'm not exactly sure what this value does. Most of the images I've seen have this value set to 0. The spec says that if there was a value specified in this byte, N, the actual ratio used would be (N + 15) / 64 for all N<>0.



Global Color Table

```
FF FF FF FF 00 00 00 00 FF 00 00 00
```

(from Sample File)

We've mentioned the global color table a few times already now let's talk about what it actually is. As you are probably already aware, each GIF has its own color palette. That is, it has a list of all the colors that can be in the image and cannot contain colors that are not in that list. The global color table is where that list of colors is stored. Each color is stored in three bytes. Each of the bytes represents an RGB color value. The first byte is the value for red (0-255), next green, then blue. The size of the global color table is determined by the value in the packed byte of the logical screen descriptor. As we mentioned before, if the value from that byte is N, then the actual number of colors stored is 2^(N+1). This means that the global color table will take up 3*2^(N+1) bytes in the stream.

Size in Logical Screen Desc	No. of Colors	Byte Length
0	2	6
1	4	12
2	8	24
3	16	48
4	32	96
5	64	192
6	128	384
7	256	768

Our sample file has a global color table size of 1. This means it holds 2^(1+1)=2^2=4 colors. We can see that it takes up 12 (3*4) bytes as expected. We read the bytes three at a time to get each of the colors. The first color is #FFFFFF (white). This value is given an index of 0. The second color is #FF0000 (red). The color with an index value of 2 is #0000FF (blue). The last color is #000000 (black). The index numbers will be important when we decode the actual image data.

Note that this block is labeled as "optional." Not every GIF has to specify a global color table. However, if the global color table flag is set to 1 in the logical screen descriptor block, the color table is then required to immediately follow that block.

Graphics Control Extension

```
21 F9 04 00 00 00 00 00
```

(from Sample File)

Graphic control extension blocks are used frequently to specify transparency settings and control animations. They are completely optional.

The first byte is the extension introducer. All extension blocks begin with 21. Next is the graphic control label, F9, which is the value that says this is a graphic control extension. Third up is the total block size in bytes. Next is a packed field. Bits 1-3 are reserved for future use. Bits 4-6 indicate disposal method. The penultimate bit is the user input flag, and the last is the transparent color flag. The delay time value follows in the next two bytes stored in the unsigned format. After that we have the transparent color index byte. Finally we have the block terminator which is always 00.

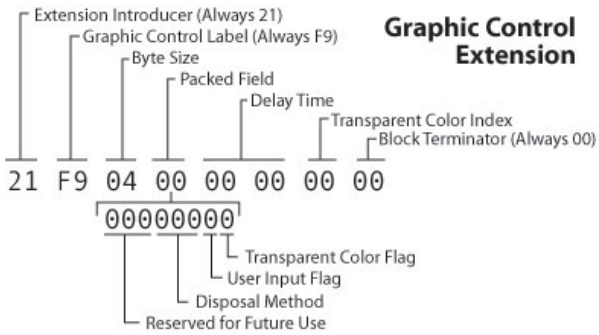


Image Descriptor

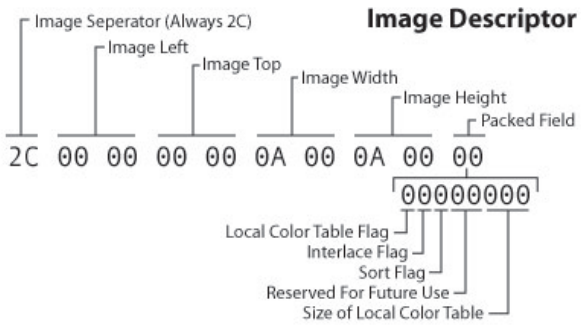
```
2C 00 00 00 00 0A 00 0A 00 00
```

(from Sample File)

A single GIF file may contain multiple images (useful when creating animated images). Each image begins with an image descriptor block. This block is exactly 10 bytes long.

The first byte is the image separator. Every image descriptor begins with the value 2C. The next 8 bytes represent the location and size of the following image. An image in the stream may not necessarily take up the entire canvas size defined by the logical screen descriptor. Therefore, the image descriptor specifies the image left position and image top position of where the image should begin on the canvas. Next it specifies the image width and image height. Each of these values is in the 2-byte, unsigned format. Our sample image indicates that the image starts at (0,0) and is 10 pixels wide by 10 pixels tall. (This image does take up the whole canvas size.)

The last byte is another packed field. In our sample file this byte is 0, so all of the sub-values will be zero. The first and most significant bit in the byte is the local color table flag. Setting this flag to 1 allows you to specify that the image data that follows uses a different color table than the global color table. (More information on the local color table follows.) The second bit is the interlace flag.



Local Color Table

The local color table looks identical to the global color table. The local color table would always immediately follow an image descriptor but will only be there if the local color table flag is set to 1. It is effective only for the block of image data that immediately follows it. If no local color table is specified, the global color table is used for the following image data.

The size of the local color table can be calculated by the value given in the image descriptor. Just like with the global color table, if the image descriptor specifies a size of N, the color table will contain 2^(N+1) colors and will take up 3*2^(N+1) bytes. The colors are specified in RGB value triplets.

Image Data

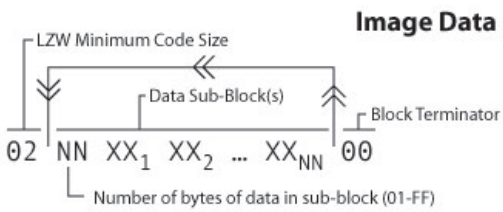
```
02 16 8C 2D 99 87 2A 1C DC
33 A0 02 75 EC 95 FA A8 DE
60 8C 04 91 4C 01 00
```

(from Sample File)

Finally we get to the actual image data. The image data is composed of a series of output codes which tell the decoder which colors to spit out to the canvas. These codes are combined into the bytes that make up the block.

The first byte of this block is the LZW minimum code size. This value is used to decode the compressed output codes. The rest of the bytes represent data sub-blocks. Data sub-blocks are groups of 1-256 bytes. The first byte in the sub-block tells you how many bytes of actual data follow. This can be a value from 0 (00) to 255 (FF). After you've read those bytes, the next byte you read will tell you how many more bytes of data follow that one. You continue to read until you reach a sub-block that says that zero bytes follow.

You can see our sample file has a LZW minimum code size of 2. The next byte tells us that 22 bytes of data follow it (16 hex = 22). After we've read those 22 bytes, we see the next value is 0. This means that no bytes follow and we have read all the data in this block.



Plain Text Extension

```
21 01 0C 00 00 00 00 64 00 64
00 14 14 01 00 0B 68 65 6C 6C
6F 20 77 6F 72 6C 64 00
```

Example (Not in Sample File)

Oddly enough the spec allows you to specify text which you wish to have rendered on the image. I followed the spec to see if any application would understand this command; but IE, Firefox, and Photoshop all failed to render the text. Rather than explaining all the bytes, I'll tell you how to recognize this block and skip over it.

The block begins with an extension introducer as all extension block types do. This value is always 21. The next byte is the plain text label. This value of 01 is used to distinguish plain text extensions from all other extensions. The next byte is the block size. This tells you how many bytes there are until the actual text data begins, or in other words, how many bytes you can now skip. The byte value will probably be 0C which means you should jump down 12 bytes. The text that follows is encoded in data sub-blocks. The block ends when you reach a sub-block of length 0.

Application Extension

```
21 FF 0B 4E 45 54 53 43
41 50 45 32 2E 30 03 01
05 00 00
```

Example (Not in Sample File)

The spec allows for application specific information to be embedded in the GIF file itself. The only reference I could find to application extensions was the NETSCAPE2.0 extension, which is used to loop an animated GIF file.

Like with all extensions, we start with 21, which is the extension introducer. Next is the extension label, which for application extensions is FF. The next value is the block size, which tells you how many bytes there are before the actual application data begins. This byte value should be 0B, which indicates 11 bytes. These 11 bytes hold two pieces of information. First is the application identifier which takes up the first 8 bytes. These bytes should contain ASCII character codes that identify to which application the extension belongs. In the case of the example above, the application identifier is "NETSCAPE" which is conveniently 8 characters long. The next three bytes are the application authentication code. The spec says these bytes can be used to "authenticate the application identifier." With the NETSCAPE2.0 extension, this value is simply a version number, "2.0", hence the extensions name. What follows is the application data broken into data sub-blocks. Like with the other extensions, the block terminates when you read a sub-block that has zero bytes of data.

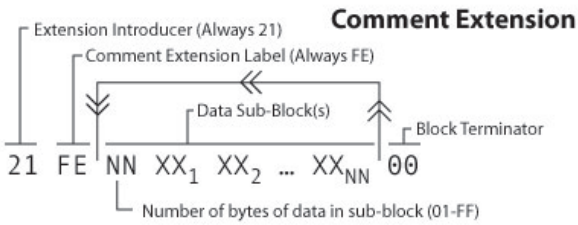
Comment Extension

```
21 FE 09 62 6C 75 65 62
65 72 72 79 00
```

Example (Not in Sample File)

One last extension type is the comment extension. Yes, you can actually embed comments within a GIF file. Why you would want to increase the file size with unprintable data, I'm not sure. Perhaps it would be a fun way to pass secret messages.

It's probably no surprise by now that the first byte is the extension introducer which is 21. The next byte is always FE, which is the comment label. Then we jump right to data sub-blocks containing ASCII character codes for your comment. As you can see from the example, we have one data sub-block that is 9 bytes long. If you translate the character codes you see that the comment is "blueberry." The final byte, 00, indicates a sub-block with zero bytes that follow, which let's us know we have reached the end of the block.



Trailer

3B (from Sample File)

The trailer block indicates when you've hit the end of the file. It is always a byte with a value of 3B.

You can read the rest of this series here:

- LZW Image Data [hn.my/imagedata]
- Animation and Transparency [hn.my/animation] ■

Matthew is currently a PhD student in the department of biostatistics at the University of Michigan where he is studying statistical genetics. He was been teaching computers new tricks ever since he started retyping BASIC programs from the back of 3-2-1 Contact magazines.

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

Our simple engagement tools help you understand your customers, prioritize feedback, and give great customer support even faster. Spend more time building a product your customers will love!

uservoice

Get 50% off your first 3 months* with the code **mindreader** at UserVoice.com.

* Offer good for new accounts if used before 12/31/2011.

Dream. Design. Print.

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

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

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

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

25% Off the First Issue You Publish

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

Please contact promo@magcloud.com with any questions.

MAGCLOUD