The Pragmatic Bookshelf

PragPub

The First Iteration

IN THIS ISSUE

* Build a one-instruction CPU with Alexander Demin
* Explore Scala pattern matching with Venkat Subramaniam
* Compare Scala and Java with Brian Tarbox
* Take the pain out of pairing with Zee Spencer



Issue #33
March 2012

Contents

FEATURES



The NOR Machine 6
by Alexander Demin

Build an assembler and an emulator for a single-instruction CPU and implement a non-trivial algorithm on it, using Ruby as a macro DSL to compile it all.



Scala for the Intrigued 25
by Venkat Subramaniam

In this seventh installment of his series on Scala, Venkat shows how Scala does pattern matching.



Comparing Java and Scala's Expressiveness 31
by Brian Tarbox

Brian converts a Java program to Scala and ponders why the Scala version is so much smaller.



Four Common Pairing Pitfalls and How to Avoid Them 35
by Zee Spencer

Pairing is hard. Here are some tips to make it easier.

DEPARTMENTS

Up Front 1
by Michael Swaine

Scala, Ruby, Java, machine architecture, pairing, pattern matching, and Daylight Savings Time.

Choice Bits 2
Drinking from the Twitter firehose.

Meet the Team 5
Meet Aron Hsaio, development editor and jack of all trades.

Calendar 38
Author sightings, upcoming conferences, and other events of note.

Shady Illuminations 43
by John Shade

John is too laid back to spring forward.

But Wait, There's More... 45
Coming attractions and where to go from here.

Except where otherwise indicated, entire contents copyright © 2012 The Pragmatic Programmers.

Feel free to distribute this magazine (in whole, and for free) to anyone you want. However, you may not sell this magazine or its content, nor extract and use more than a paragraph of content in some other publication without our permission.

Published monthly in PDF, mobi, and epub formats by The Pragmatic Programmers, LLC, Dallas, TX, and Raleigh, NC. E-Mail support@pragprog.com, phone +1-800-699-7764. The editor is Michael Swaine (michael@pragprog.com). Visit us at http://pragprog.com for the lowdown on our books, screencasts, training, forums, and more.

ISSN: 1948-3562

Up Front

Ruby as Assembly Language

by Michael Swaine



The Simplest Thing That Could Possibly Work

What's the smallest instruction set that a functional computer could have? The answer is: one instruction. You can build all of logic and computation from combinations a single NOR gate. But would you be crazy enough to do it? Alexander Demin decided the answer was yes, and he shares his experience—and his code—in this issue. Follow along as he builds an assembler and an emulator for a single-instruction CPU and implements a non-trivial algorithm on it, using Ruby as a macro DSL to compile it all.

Apart from that adventure in Ruby-as-assembly-language, we're highlighting Scala again this issue, with two articles that approach the language very differently. Scala expert Venkat Subramaniam (Programming Scala [U1]) continues his tutorial series on the language, looking at Scala's capabilities for pattern matching. And Brian Tarbox, an unabashed Scala newbie, shares his experience in porting a Java program to Scala.

To Zee Spencer, pair programming is a powerful tool, and you know what can happen if you don't use power tools carefully. Zee shares four antipatterns of pairing that he's figured out over the years. Avoid these and this tool will be less likely to bite you.

Also in this issue, John Shade takes on the government and we introduce our latest Pragmatic team member. We hope you enjoy it.

Solution to Last Issue's Quiz

Last month's quiz was a really easy cryptogram puzzle. The encrypted words in the first sentence, we said, could be rearranged to form the encrypted words in the second sentence. We asked you to decrypt the two sentences. The sentences were:

"How long before BKGK GMHWDFMB Google for search?" and "The only book in the dining car was GDKWB GMFKHMB."

Which, decrypted, become:

"How long before SIRI REPLACES Google for search?" and "The only book in the dining car was RAILS RECIPES."

Download *PragPub* to Your Kindle Automatically

Thanks to Jason Young, we have an IFTTT [U2] recipe to download the latest issue of *PragPub* to your Kindle when I tweet its announcement. If I didn't mess it up, this recipe [U3] should do the trick.

Choice Bits

Time Travel, Cow Belching, and Other Timely Topics

Plumbing the depths of the twitterstream and keeping our feet dry.



What's Hot

Top-Ten lists are passé—ours goes to 11. These are the top titles that folks are interested in currently, along with their rank from last month. This is based solely on direct sales from our online store.

1	1	Agile Web Development with Rails
2	NEW	The Developer's Code
3	5	The Cucumber Book
4	9	Programming Ruby 1.9
5	2	Deploying Rails
6	10	The RSpec Book
7	8	Seven Databases in Seven Weeks
8	NEW	Programming Your Home
9	6	iOS SDK Development
10	11	Rails Recipes
11	NEW	CoffeeScript

Things Inquiring Minds Want to Know

- *Dear logitech, love my wave wireless keyboard. Don't need the numeric keypad. How can I safely saw it off? tx, Kent.* — @KentBeck
- *Does anybody know the name of, source of, maker of, the urinal in Greens Restaurant in Ft. Mason, Frisco? This is not a joke.* — @MrAlanCooper
- *Why didn't they make throat armor in the Game of Thrones days? There's a blacksmith out there that should be fired.* — @SarahKSilverman
- *Wait—is Moneyball nominated for Best Spreadsheet or for Fewest Women?* — @ftrain

Thanks for Sharing That

- *Girl eats 4,000 washing-up sponges. bit.ly/yNKIAh /me: posted without comment.* — @scottdavis99
- *Dawn is off to teach eager veterinary students about cow belching. [I don't make the news; I just report it.]* — @marick
- *Japanese Fart Scrolls. A whole new sphere of academic study. tofugu.com/2012/02/18/jap…* — @stephenfry
- *US crackdown on crystal meth means Sudafed is hard to buy. But meth is easy to buy; here's how to make Sudafed from it. heterodoxy.cc/meowdocs/pseud…* — @ciphergoth

Fame & Fortune

- *I tried to start my own political party once, but we failed to gain support. http://sgp.cm/0757d1* — @cmdrtaco
- *I would gladly trade all my fame and fortune for even more fame and fortune.* — @ConanOBrien
- *Yelp IPO: 8 yr-old company, $83M in revs (up 70% y/y), $16M in loss (up 100% y/y), valued at $1.3B. You do the math.* — @dhh
- *I am told I am trending worldwide on twitter—but I feel the same here stuck in traffic.* — @johncusack
- *No Tweet results for yelp fame fortune.* — @twitter

Time Travail

- *OMG this is seriously good—the new Manchester Time Machine app from the @NWfilmarchive. s.coop/815c* — @MancLibraries
- *Someone tweeted me today asking if I could check out their demos on MySpace… proof that time travel is possible.* — @example
- *Dear future self: if you're in a bad mood, just remember what today felt like.* — @ginatrapani
- *I always give my future self way too much credit when it comes to remembering passwords.* — @davepeck
- *Facebook's future, 2012-2025 [infographic] is.gd/S4qkEa* — @GuyKawasaki
- *Facebook is our past. Twitter is our present. Unemployment is our future.* — @MoistPork
- *I think @robcorddry will be nominated for hot tub time machine at some point in the future and past.* — @johncusack

Gripes, Grovels, Rants, and Regrets

- *Berners-Lee "sorry" for HTTP's forward slashes. blork.ly/y1dwDl* — @davewiner
- *Today I earned my black belt in Groveling for Conference Rooms.* — @noelrap
- *When you say "You missed my point," you're blaming the wrong person.* — @petdance
- *I'd rather tell my invoice resource it is now paid, rather than that its "paid" attribute became 1.* — @peterc
- *Let's argue about whether Haskell or Clojure is better while somebody else ships products using PHP and duct tape.* — @agentdero
- *I had resolved to stay out of arguments on the Internet, but that's shaping up to be too drastic a lifestyle change.* — @lizardbill

This Coding Life

- *A bunch of the cool kids in the Ruby/Rails world use vi. Looks like tmux is your friend. Watch youtu.be/JXwS7z6Dqic buy pragprog.com/book/bhtmux/tm…* — @pragdave
- *From page 35 of the English:Ruby phrase book: "I hate you with a seething passion" translates roughly as "rescue nil."* — @cowboyd
- *Ruby 1.9 has support for GOTO through the SUPPORT_JOKE compiler flag. I am not making this up. patshaughnessy.net/2012/2/29/the-…* — @ReinH
- *One assertion per test is an excellent way to exercise the reader's short term memory and the fine motor skill of scrolling.* — @marick
- *My phone is telling me that the high today is 48, the low is 50, and it's currently 37. Related: computers control markets and fly planes.* — @Chris_Ashworth

- *Beginning to think that the software that cracks CAPTCHAS is technically passing the Turing test.* — @hal_hildebrand
- *Reclaim the internet and your life by using #IFTTT. youtu.be/cAbB6qmTFU4* — @ParkerXL
- *The Anti-IF Campaign is recruiting new members. Join us! bit.ly/2qyIk* — @AntiIFSchool
- *Implementation and Big Thoughts are the chocolate and peanut butter of programming.* — @KentBeck
- *"The real problem is that since you are in Australia, all bits must be turned upside-down during transmission."* — @codinghorror
- *Some programmers, when confronted with a problem, think "I know, I'll use floating point arithmetic." Now they have 1.999999999997 problems.* — @tomscott
- *When OS X *fully* embraces iOS, that'll be the Cougar release.* — @JohnShade1
- *Programming as brussel sprouts: geekymomblog.com/2012/02/24/is-…* — @gvwilson

Observations

- *Listen first, know your friends, nothing is private, stay on guard, don't feed trolls, create more than you consume: j.mp/AwZZji* — @rands
- *Sacred cows make the best hamburger. - Mark Twain* — @bakadesuyo
- *RT @joshlam: Google was late 2 search. Facebook was late 2 social networking. Apple was late 2 MP3 player. Late is nothing. Just do it better.* — @MrAlanCooper
- *Scientific goat breeding program + fence vs horny buck. Bet on the buck. Every time.* — @KentBeck
- *A pizza with radius "z" and thickness "a" has the volume pi*z*z*a. #math #education* — @intmath
- *The Bard retwitten? #SeussSpeare, I'm smitten.* — @pragpub
- *"What you wake up on Saturday thinking about is your passion and what u should do." -Reid Hoffman* — @puneetchawla
- *Getting an initial contact from another company through their SEO firm is like getting a love letter from your girlfriend's drug dealer.* — @peterc
- *The phrase "Don't take this the wrong way" has a zero percent success rate.* — @nealbrennan
- *If every claim of "X produces Y" included a Tukey bar-and-whisker plot, humanity would be better off.* — @marick
- *If dolphins were as smart as people say, they'd stop hanging around with tuna fish.* — @MrAlanCooper

Who Are Those Guys? [U1]

They're not all guys. Also, we have to confess that we cleaned up their punctuation and stuff a little. OK, who they are: Anti-IF School, Chris Ashworth, Jeff Atwood, Eric Barker, Kent Beck, Murray Bourne, Neal Brennan, Puneet Chawla, Alan Cooper, Peter Cooper, Paul Crowley, R. Tyler Croy, John Cusack, Scott Davis, example, Paul Ford, Stephen Fry, David Heinemeier Hanssen, Rein Henrichs, Hal Hildebrand, Guy Kawasaki, Andy Lester, Manchester Libraries, Bill the Lizard, Charles Lowell, Rob Malda, Brian Marick, MoistPork, Conan O'Brien, Parker, Dave Peck, rands, Noel Rappin, Tom Scott, John Shade, Sarah Silverman, Michael Swaine, Dave Thomas, Gina Trapani, Twitter, Greg Wilson, and Dave Winer. You can follow us at www.twitter.com/pragpub [U2].

Meet the Team

Development Editor Aron Hsaio

Each month we profile a member of the Pragmatic Bookshelf team. This month it's development editor Aron Hsaio, our newest recruit.





Aron Hsaio, Development Editor

Aron is a recent addition to the Pragmatic team and something of jack of all trades. Throughout the 1990s and 2000s Aron wrote a number of books on Linux and open source platforms and worked as an editor for technology publishers like Sams and Que. More recently he lent his talents to companies and interests as diverse as policy think tanks like Media Tenor International and the Annenberg Center for Global Communication Studies, web properties like eBay.com and About.com, and universities like NYU and The New School.

On a typical day, Aron spends a few hours as a webmaster kludging up tools for CMS backends, a few hours as a classroom instructor teaching core courses for sociology or media studies undergrads, a few hours as a researcher going over data and journals, and a few hours as an editor working on Pragmatic titles and projects. Afterward, he goes home to spend a few much-needed hours being a simple husband and father.

Aron considers himself to be an enthusiast both of technology and of society. In addition to being longtime advocate of user-centric approaches in his computing and IT roles, Aron is also a strong advocate of technology-centric understandings of human behavior in social research, seeing technology as fundamental to everyday patterns of human life and basic human beliefs about the world, no matter the historical period.

Though he started this journey as a 14-year-old computer science major with a knack for networks and K&R C, Aron went on to earn degrees in cultural anthropology and interdisciplinary social research. He is currently working to finish his Ph.D. in the sociology of media and technology, but his progress is continually slowed by his impulsive needs to play at being a stock photographer, to read travelogues and Robert Bolano novels, to go on extended road trips, and to keep incredibly needy aquarium fish.

Email: aron@pragprog.com [U1]

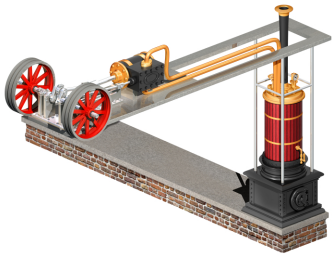
Personal Website: Leapdragon 2012 [U2]

The NOR Machine

Build a CPU with Only One Instruction

by Alexander Demin

Build an assembler and an emulator for a single-instruction CPU and implement a non-trivial algorithm on it, using Ruby as a macro DSL to compile it all.



Have you ever developed in an assembly language? Have you *developed* an assembly language? Ever developed a *CPU* running your own assembly language?

There may be some who said "yes" to all three questions. But even for those readers, this article may have something new to offer. We are going to:

1. Develop a CPU that performs only one primitive instruction.
2. Write a non-trivial calculation in this CPU's single-instruction assembly language.
3. Use Ruby as a macro domain-specific language to compile it all.

At the end we will have an assembler and an emulator of that single instruction CPU, along with an implementation of a non-trivial algorithm on it, and some Ruby code that looks like no Ruby code you've ever seen.

Why would we want to do this? What, doesn't it sound like fun?

One Function to Build Them All

I picked up an idea years ago in a FIDO discussion group, RU.HACKER. With a minor improvement, that idea will become the heart of our CPU.

The idea is to use just one function to compute everything. It suffices to do this for bits: bytes consist of bits, and if we can compute bits, we also can compute bytes, and everything made up of bytes. That is, everything.

There are sixteen boolean functions of two arguments. Two of them are special: NOR and NAND. The other fourteen functions can be computed using only either NOR or NAND. We'll use NOR.

Not surprisingly, NOR can be expressed via other functions:

```
NOR(a, b) = NOT(OR(a, b))
```

NOR's truth table is as follows:

```
===== ========
Input  Output
===== ========
 A B  A NOR B
----- --------
 0 0     1
 0 1     0
 1 0     0
 1 1     0
===== ========
```

Remember, all other boolean functions of one or two arguments can be computed from NOR. For example:

```
NOT(a) = NOR(a, a)
AND(a, b) = NOT(OR(NOT(a), NOT(b)))
OR(a, b) = NOT(NOR(a, b))
XOR(a, b) = OR(AND(a, NOT(b)), AND(NOT(a), b)))
```

Defining Our CPU's Architecture

So NOR gives us everything we need to compute anything in bits. All right, we have established our single function. Now let's sketch out an overall architecture for our CPU.

We will have a linear memory organization containing 2^16 (65536 or 0xFFFF), cells, and consequently a range of valid addresses from 0 (0x0000) to 65535 (0xFFFF), 16 bits per address. So far this CPU is similar to classical microprocessors like the Intel 8080 or Zilog Z80. But from here on it's going to get really different.

The first significant difference is that memory cells in our CPU will be 16-bit, not 8-bit, so one cell can hold a range of values from 0x0000 to 0xFFFF.

The next difference is more important. Normally, in both RISC and CISC architectures of modern CPUs, each individual instruction consists of two parts: an operation code, or *opcode*, and some number of *arguments*. The opcode specifies an operation to perform and the arguments define the details of a particular instruction.

But remember, our CPU understands only one instruction. So we don't need the opcode, because we always compute NOR! Our instructions will only contain the arguments—the arguments of NOR.

Now we have to settle on a binary format for our instructions. Each instruction will reside in three consecutive cells in the memory. The first and second cells will contain addresses of the NOR arguments, a and b respectively. The third argument will contain the address where the result of the NOR will be placed.

Imagine our memory as a linear array mem of 16-bit values:

```
mem[0], mem[1], mem[2], mem[3], ..., mem[65534], mem[65535]
```

Consider an instruction at the address 1. It occupies three cells: 1, 2, and 3. To process this instruction, our CPU has to execute the following:

1. take a value from mem[1] and name it addr_a
2. take a value from mem[addr_a] and name it a
3. take a value from mem[2] and name it addr_b
4. take a value from mem[addr_b] and name it b
5. compute r = NOR(a, b)
6. take a value from mem[3] and name it addr_r
7. put r to mem[addr_r]

We can generalize this by introducing a register called IP (we'll consider registers in a minute), an instruction pointer, holding an address of the current instruction. Let's write down the algorithm in a pseudo code:

```
mem[mem[IP + 2]] = NOR(mem[mem[IP + 0]], mem[mem[IP + 1]])
```

Once an instruction is processed, we increment IP by 3 and repeat again. You may notice that this loop is infinite, similar to a real CPU. Of course, when we develop an emulator for our CPU we will define a special case when the CPU halts.

Creating Registers

Okay, we have the memory model and the instruction format. Now we need to think about registers. In real CPUs the registers are a bunch of cells usually residing on the main chip and can be read or written very quickly. These cells are supposed to contain data that will be accessed frequently.

Our CPU is not real hardware and we don't have that fast on-chip memory for registers, so we'll just put the registers of our CPU into the main memory (from now on, just memory), in fact, into the mem array. This is a very interesting trick. We'll see why in a second.

We have already declared one register, called IP, holding an address of the current instruction. Because our registers are in the memory, the IP register is also in the memory. We said the instructions in our CPU get data from two cells, calculate NOR, and put a result into another cell. Our IP register is a regular cell in the memory, which means it can be changed similarly to other cells by any instruction. This is how we're going to implement branching.

Looking ahead, you could already guess that our CPU holds the registers, the code, and data in the same memory (the mem array).

Everything is ready to implement the main loop of our CPU. Let's do it in Ruby:

```
def nor(a, b)
  ~(a | b) & 0xFFFF
end
while true do
  i = mem[IP]
  a = mem[i + 0]
  b = mem[i + 1]
  r = mem[i + 2]
  mem[ip] = i + 3
  f = nor(mem[a], mem[b])
  mem[r] = f
end
```

A Ruby IP variable specifies an address on the IP register in the mem array.

Also, you have probably noticed that a function nor(a, b) doesn't do just a single 1-bit NOR operation, but it computes sixteen NORs in parallel (remember, our registers and memory cells are 16-bit): bit-wise NOR between a and b.

Creating More Instructions

All right, our CPU is almost ready. Now we need to learn how to write programs on it. We can compute NOR and other boolean functions. But this

is not enough. We need to implement data copying, branching, and multi-bit arithmetic.

At this point it is worthwhile to introduce a slightly different notation for our NOR-based assembly language. Previously we used a notation of functions (for example, NOT(a) = NOR(a, a)), but let's change it to an assembly language.

Our primitive instruction NOR will have following syntax:

```
NOR a, b, r
```

The first word in the line is an instruction name (NOR) and then arguments follow.

Let's rewrite other functions into this notation. Also, for compound functions, we will use a pseudo macro assembler syntax.

NOT, defined as NOT(a) = NOR(a, a):

```
macro NOT a, r
  NOR a, a, r
endm
```

More complex operations require temporary registers. We will declare them using local keywords. These registers are similar to IP in that they reside somewhere in the memory.

AND, defined as AND(a, b) = NOT(OR(NOT(a), NOT(b))):

```
macro AND a, b, r
  local t1, t2
  NOT a, t1
  NOT b, t2
  OR t1, t2, t1
  NOT t1, r
endm
```

OR, defined as OR(a, b) = NOT(NOR(a, b)):

```
macro OR a, b, r
  local t
  NOR a, b, t
  NOT t, r
endm
```

XOR, defined as XOR(a, b) = OR(AND(a, NOT(b)), AND(NOT(a), b))):

```
macro XOR a, b, r
  local t1, t2
  NOT b, t1
  AND a, t1, t1
  NOT a, t2
  AND t2, b, t2
  OR t1, t2, r
endm
```

A MOV command, copying data from one place to another, can be implemented relatively easily. For example, we want to copy a word from a from address to a to address:

```
mem[to] = OR(mem[from], mem[from])
```

or in the form of a macro:

```
macro MOV from, to
  OR from, from, to
endm
```

A JMP command organically follows from MOV. If the to argument of MOV is the IP register, we effectively write a value of to to IP and make the CPU execute the next instruction from another address, from:

```
mem[IP] = OR(mem[from], mem[from])
```

or in the form of a macro:

```
macro JMP to_addr
  OR to_addr, to_addr, IP
endm
```

Conditional branching is more interesting. Let's set a condition parameter to be 0xFFFF if the condition is true and we do a jump, and 0x0000 otherwise. So the conditional branching can be implemented as:

```
mem[IP] = OR(AND(to_addr, condition), AND(mem[IP], NOT(condition)))
```

or as a macro:

```
macro BRANCH true_addr, false_addr, condition
  local t1, t2
  AND true_addr, condition, t1
  NOT condition, t2
  AND false_addr, t2, t2
  OR t1, t2, IP
endm
```

If condition is 0xFFFF, we write true_addr to IP and jump, but if condition is 0x0000, we write false_addr to IP. Writing to IP means that on the next CPU cycle the execution will continue from another address.

Implementing Basic Math

Let's pause for a second and think what we've done so far.

Having only one instruction, NOR, we have built NOT, AND, OR, XOR, MOV, JMP, and BRANCH. So we now know how to do boolean computations, copying data, and branching. Isn't it cool?

Agreed, we lack multi-bit arithmetic. We really only need summation. With summation we can build subtraction, multiplication, and division. And once we have all of these, we can program any algorithm on our CPU.

A very important point is that we need 16-bit summation. Remember, one instruction in our CPU computes bit-wise NOR. This means that one bit from the a argument and its corresponding bit from b affect only one resulting bit of r. But an algorithm of the full binary adder entails the concept of carry, where we have to carry an extra bit to the summation of the next binary digit.

Currently we cannot move bits from one position to another. We have no shift operation. Let's improve the main CPU loop.

We introduce one extra register and call it SHIFT_REG. This register will contain the result of the last NOR operation cyclicaly shifted to the left.

```
while true do
  i = mem[IP]
  a = mem[i + 0]
  b = mem[i + 1]
  r = mem[i + 2]
  mem[ip] = i + 3
  f = nor(mem[a], mem[b])
  mem[r] = f
  mem[SHIFT_REG] = ((f >> 15) & 1) | ((f & 0x7FFF) << 1)
end
```

The last line computes a cyclic left shift of the last NOR and stores the result to the SHIFT_REG register.

Now let's build a full binary adder.

The full binary adder has the following logic (suppose that a and b are bits):

```
sum = (a ^ b) ^ carry
carry = (a & b) | (carry & (a ^ b))
```

Here a and b are arguments of summation and carry equals 1 if the previous summation had an overflow and we need to take this fact into account.

Initially carry equals 0, and after computing sum we also update carry.

Below is an implementation of the full binary adder. This is a macro based on existing primitive functions (XOR, AND, OR and MOVE).

```
; Input:
;  mask  - a current bit mask (0x0001, 0x0002, 0x0004, 0x0008 etc)
;          This parameter specifies which bit of "a" and "b" we are
;          going to add.
;  carry - a carry from the previous summation
;          This values is "masked" (ANDed) by the "mask" argument.
;  a, b  - an argument addresses
;  r     - an address of the result
; Output:
;  r     - a result
;  carry - a carry to the next bit (already shifted to the left
;          with respect to the "mask")
;  mask  - a mask shifted to the left by one bit
;
macro FADD mask, carry, a, b, r
  local t1, t2, bit_r       ; Local variables
  XOR a, b, t1              ; Formula: sum = (a ^ b) ^ carry.
  XOR t1, carry, bit_r      ;
  AND bit_r, mask, bit_r    ; Mask all bits in 'bit_r'
                            ; except the current one
  OR bit_r, r, r            ; Save the bit to the result: r |= sum
  AND a, b, t2              ; Formula:
                            ; carry = (a & b) | (carry & (a ^ b))
  AND carry, t1, t1        ;
  OR t2, t1, carry          ; The carry is calculated. Its left
                            ; shifted copy is in SHIFT_REG.
  MOV SHIFT_REG, carry      ; (1): Assign the carry to itself but
                            ;      shifted the next bit.
  MOV mask, mask, mask      ; (2): Dummy assignment to just get:
                            ;      SHIFT_REG = mask << 1.
  MOV SHIFT_REG, mask       ; (3): mask = SHIFT_REG = mask << 1
  AND carry, mask, carry    ; Mask all bits in 'carry'
                            ; except the current one
endm
```

A trick occurs at the line labeled (1). On the line previous to (1) we update carry and after that OR operation, the SHIFT_REG register holds a value of carry shifted to the left. At line (1) we copy that shifted value of carry to carry itself.

Another interesting trick is at lines (2) and (3). At line (2) we do a "weird" operation MOV mask, mask, mask which seems to be doing nothing at all and definitely doesn't change mask. But behind the scenes, the MOV operation also shifts the value of mask to the left and stores the result into S. Then on line (3) we copy that shifted value back to mask. By doing those lines (2) and (3) we prepare mask for summation of the next bit. Finally we clean up carry by zeroing to all except the current, pointed to by mask.

Okay, are you ready for the full 16-bit adder? Here we go.

To make the source shorter we need to introduce one more feature to our pseudo macro assembler: a construct to repeat lines. Its syntax will be *[number]. If a line has such prefix, for example, *16, this line will be repeated a [number] of times.

Now the full 16-bit adder macro:

```
; Input:
;  a, b  - arguments
;  carry - a carry (the least significant bit only matters)
; Output:
;  r     - the result: r = a + c + carry
;  carry - a carry (the least significant bit only matters)
;
macro ADC a, b, carry, r
  local mask                      ; Local variables.
  XOR r, r, r                     ; Set r = 0
  MOV CONST_1,  mask              ; The initial mask value = 0x0001.
  *16 FADD mask, carry, a, b, t   ; Repeat FADD 16 times (no loops,
                                  ; just a repetition).
  MOV t, r                        ; Move the result 'r'.
endm
```

We have called the macro ADC which stands for "Addition with Carry."

We initialize a result to 0 and then put 0x0001 to mask. We introduce a special cell called CONST_1. This cell simply contains a value of 0x0001. Then we repeat the FADD macro sixteen times. Remember, on each step the FADD updates carry and mask for the next iteration. After sixteen FADD repetitions the carry has been cyclicaly shifted from the most significant sixteenth bit to the lowest significant first bit.

That is it. The full 16-bit summation with carry is ready.

Programming a Real Algorithm

Let me recap once again what we have so far. We have 64K memory of 16-bit values and two registers, IP and SHIFT_REG. We can do boolean functions, we can copy cells and do branching, and finally we have 16-bit summation with carry.

Now we have enough to program any algorithm on our CPU.

You may say we lack the call stack and the concept of subroutines. Do you want to implement them? Let's implement two auxiliary operations for indirect data access first.

We call them PEEK and POKE, similar to operators in the BASIC language, where the first one returns a value from a given address and the second one puts a value to a given address. Implementations of these operations are interesting because we will write code modifying itself on the fly. This is possible because the registers, code, and data in our CPU share the same address space, the memory.

Okay, let's do PEEK first.

```
; This macro reads a value from the address given in "a"
; and stores it to the cell whose address is "b."
;
; Input:
;   a - an address
; Output
;   b - a values from a cell addressed by "a"
;
; This is an indirect addressing: [a] -> b
;
macro PEEK a, b
  local t
  MOV a, label1  ; In these two lines we create
  MOV a, label2  ; a "NOT x, y, t" command on-the-fly.
 label1:         ; Initially there is a "NOT 0, 0, t"
  dw 0           ; command here but by two previous "MOV"s
 label2:         ; we replace two zeros from with
  dw 0           ; a concrete value of "a"
  dw t
  NOT t, b       ; Final NOT to "b".
endm
```

Technically, PEEK is similar to MOV. It does two subsequent NOTs to copy a value from one cell to another. The trick is that we modify the first NOT by injecting the concrete address a. The second NOT is a regular one. The label1 and label2 are local for this macro.

Let's implement POKE.

```
; This macro writes a given value "a"
; to a cell addressed by "b."
;
; Input:
;   a - a value
;   b - an address
; Output:
;   None.
;
; This is an indirect addressing: a -> [b]
;
macro POKE a, b
  local t
  MOV b, label    ; We create a "NOT t, t, x" command on-the-fly.
  NOT a,  t       ; This is a regular "NOT" operation.
  dw @t           ; This is a "NOT t, t, 0" command
  dw @t           ; and by the first "MOV" in the macro
 label:           ; we replace that "0" with a value of "b"
  dw 0
endm
```

POKE also uses the MOV technique. It has two subsequent NOT operations and the second one is being modified on the fly by injecting the concrete address of b.

How cool is that?!

PEEK and POKE are very powerful operations. Having them, it is very easy to implement a concept of the call stack to be able to call subroutines. I leave it as an exercise for the readers.

Let's make our CPU compute something useful. For example, CRC16.

The algorithm in Ruby might look like:

```
def calc_crc16(data)
  crc = 0xFFFF
  data.each_byte do |x|
    crc = crc ^ (x & 0xFF)
    8.times do
      shift = (crc & 1) != 0
      crc = crc >> 1
      crc = crc ^ 0x8401 if shift
    end
  end
  return crc
end
```

Usually when people develop in an assembly language they use a translator, an assembler, converting a text representation of CPU instructions into machine code. Also the assembler can do macro substitution (we have already used macros, and we do need this functionality), data allocation, and symbolic labels.

The assembler in most cases is a standalone application that takes a source in the assembly language and produces a binary.

To make our stuff a bit more fun we'll use another approach. We will use Ruby as our assembler. We will develop a few helper functions, allowing us to write regular Ruby code, and the code will automatically produce our NOR-based machine code. You'll be amazed at how great Ruby is at doing that.

And finally we'll run a compiled binary on the emulator.

Let's begin not from the low-level helpers, but from the CRC implementation.

The code below computes the CRC-16 from a string given in a needle variable. I know, some macros and commands may look new to you, but we'll consider them shortly.

Just reflect on this code for a second. Does it look like Ruby at all? To me it looks like assembly language. Here you can unfold the magic of Ruby because it is Ruby. The tiny helper functions make it possible.

```
needle = literal Example + "\x00"
i, ch, t, ptr, crc16 = var 5
label :entry
      MOVi 0xFFFF, crc16
      MOVi needle, ptr
label :crc_loop                  # crc_loop:
      PEEK ptr, ch               #   ch = *ptr
      IS_0 ch                    #   is ch == 0?
      JZi  :exit                 #   if yes, goto "exit"
      ANDi ch, 0xFF, ch          #   ch &= 0xFF
      XOR  crc16, ch, crc16      #   crc16 ^= ch
      MOVi 8, i
label :loop_i                    # crc_loop_i:
      ANDi crc16, 1, t           #   t = crc16 & 1
      SHR crc16, crc16           #   crc16 >>= 1
      IS_0 t                     #   is t == 0?
      JZi :skip_xor              #   if yes, goto "skip_xor"
      XORi crc16, 0x8401, crc16  #   crc16 ^= 0x8401
label :skip_xor                  # skip_xor:
      ADDi i, 0xFFFF, i          #   i -= 1
      IS_0 i                     #   is i == 0?
      JNZi :loop_i               #   if not, goto "loop_i"
      ADDi ptr, 0x0001, ptr      #   ptr += 1
      JMPi :crc_loop             #   goto crc_loop
      label :exit
      EXITi()
```

The Full Source

All right, now we are approaching the finish line. We'll go through the entire source of the program in Ruby, which contains this CRC-16 calculation code, the helper functions, NOR-based binary generation routines, and, finally, the emulator of our CPU.

When this code runs, it calculates CRC-16 twice, using our NOR CPU and a regular Ruby implementation. I hope the results will match!

Line 1 sets up a string we use to calculate CRC-16.

```
1  Example = "String for testing"
```

From lines 2 to 37 there are helper functions allowing us to make Ruby code look so close to the assembly. We have functions:

- to declare and set symbolic labels (make_label() in line 8 and label(), line 14)
- to place NOR code into the assembly (code() in line 15)
- to place a string into the assembly (literal() in line 16)
- to declare and place into the assembly different types of variables and constants (static() in line 21, var() in line 24, init_var() in line 29, and const in line 34)

```
 2  Text = []
 3  Labels = []
 4  Statics = {}
 5  Consts = {}
 6  Vars = {}
 7  Literals = {}
 8  def make_label(name = "")
 9    name = name.to_s
10    name = "label_%d" % Labels.size if name.empty?
11    Labels.push name
12    return name
13  end
14  def label(name) Text.push "#" + name.to_s end
15  def code(value) Text.push "  %s" % value end
16  def literal(value)
17    name = "str_%d" % Literals.size
18    Literals[name] = value
19    return name
20  end
21  def static(*names)
22    names.each { |name| Statics[name] = 0 }; names
23  end
24  def var(count = 1)
25    names = []
26    count.times { Vars[(names.push "var_%d" % Vars.size).last] = 0 }
27    return names
28  end
29  def init_var(value)
30    name = "var_%d" % Vars.size
31    Vars[name] = value
32    return name
33  end
34  def const(value)
35    return Consts[value] if Consts.has_key? value
36    Consts[value] = "const_%d" % Consts.size
37  end
```

In lines 38-39 we define the main registers of our CPU: IP, SHIFT_REG, CARRY_REG, and ZERO_REG.

```
38  IP = init_var :entry
39  SHIFT_REG, CARRY_REG, ZERO_REG = var 3
```

The function NOR() in line 40 is our main building block. All other functions are built using this one. This function places into the assembly three words of the instruction.

```
40  def NOR(a, b, r)
41    code a
42    code b
43    code r
44  end
```

In lines 45-89 we define the bit-wise boolean functions jump and move. Their implementations repeat exactly what we've discussed previously. The static helper function allows us to define temporary variables for each macro. These temporary variable are singletons. There is one instance of each variable (for example, OR_t) across all invocation of a macro where it is defined.

It is worthwhile to look at functions having "i" in the name (ANDi() in line 60, XORi() in line 73, JMPi() in line 84 and MOVi() in line 87). These functions have an immediate value in the second parameter. Normally all arguments of our

macros are addresses pointing to the values, not values per se. Each of these functions has to place the immediate value to a temporary variable first and then use the const() helper to create an unnamed constant.

```
45  def NOT(a, r)
46    NOR a, a, r
47  end
48  def OR(a, b, r)
49    t = static :OR_t
50    NOR a, b, t
51    NOT t, r
52  end
53  def AND(a, b, r)
54    t1, t2 = static :AND_t1, :AND_t2
55    NOT a, t1
56    NOT b, t2
57    OR t1, t2, t1
58    NOT t1, r
59  end
60  def ANDi(a, imm, r)
61    t = static :ANDi_t
62    MOVi imm, t
63    AND a, t, r
64  end
65  def XOR(a, b, r)
66    t1, t2 = static :XOR_t1, :XOR_t2
67    NOT a, t1
68    NOT b, t2
69    AND a, t2, t2
70    AND b, t1, t1
71    OR t1, t2, r
72  end
73  def XORi(a, imm, r)
74    t = static :XORi_t
75    MOVi imm, t
76    XOR a, t, r
77  end
78  def MOV(a, b)
79    OR a, a, b
80  end
81  def JMP(a)
82    MOV a, IP
83  end
84  def JMPi(a)
85    JMP const(a)
86  end
87  def MOVi(imm, a)
88    MOV const(imm), a
89  end
```

In lines 90-105 there is an implementation of a PEEK() function. As we have discussed previously this function applying indirect addressing has to modify its code on the fly by injecting the a value to the second NOR (line 97-103). We create two labels l1 and l2 (lines 92 and 93) and use them (lines 98 and 100) to address the locations where a has to be injected.

```
 90  # [a] -> b
 91  def PEEK(a, b)
 92    l1 = make_label
 93    l2 = make_label
 94    MOV a, l1
 95    MOV a, l2
 96    t = static :PEEK_t
 97    # BEGIN: NOR 0, 0, t
 98  label l1
 99    code 0   # <- a [initially '0']
100  label l2
101    code 0   # <- a [initialli '0']
102    code t
104    NOT t, b
105  end
```

In lines 106-118 there is a POKE() function. This function also does indirect addressing; that's why it modifies itself similar to PEEK(). We create a label l in line 108 and place it in line 115. The label l identifies a location to inject the value of b.

```
106  # a -> [b]
107  def POKE(a, b)
108    l = make_label
109    MOV b, l
110    t = static :POKE_t
111    NOT a, t
112    # BEGIN: NOR t, t, 0
113    code t
114    code t
115  label l
116    code 0   # <- b [initially '0']
118  end
```

In lines 119-121 there is a function halting our CPU. We haven't discussed yet how to stop the CPU. Below is one possible way. Say we stop the CPU when IP becomes equal to 0xFFFF. Of course, the emulator needs to implement this behavior, and you'll see it further down.

```
119  def EXITi
120    MOVi 0xFFFF, IP
121  end
```

The Full Binary Adder

In lines 122-149 there is an implementation of the full binary adder. The function FADD() does what we discussed, and the implementation is nicely wrapped into Ruby helpers. Note that this function doesn't use the CARRY_REG register directly (as with SHIFT_REG). Instead it uses the carry argument passed from a caller function.

```
122  def FADD(mask, carry, a, b, r)
123    tmp_a, tmp_b, bit_r = static :FADD_a, :FADD_b, :FADD_bit_r
124    t1, t2 = static :FADD_t1, :FADD_t2
125
126    AND a, mask, tmp_a      # zeroing bits in 'a' except mask'ed
127    AND b, mask, tmp_b      # zeroing bits in 'b' except mask'ed
128    AND carry, mask, carry  # zeroing bits in 'carry' except mask'ed
129    # SUM = (a ^ b) ^ carry
130    XOR a, b, t1
131    XOR t1, carry, bit_r
132    # Leave only 'mask'ed bit in bit_r.
133    AND bit_r, mask, bit_r
134    # Add current added bit to the result.
135    OR bit_r, r, r
136    # carry = (a & b) | (carry & (a ^ b))
137    AND a, b, t2
138    AND carry, t1, t1
139    # carry is calculated, and 'SHIFT_REG' contains the same value
140    # but shifted to the left by 1 bit.
141    OR t2, t1, carry
142    # CARRY is shifted to the left by 1 bit for the next round.
143    MOV SHIFT_REG, carry
144    # SHIFT_REG = mask << 1
145    MOV mask, mask
146    # mask = shift (in fact, "mask = mask << 1")
147    MOV SHIFT_REG, mask
148    AND carry, mask, carry
149  end
```

A function ZERO() in line 150-152 uses the XOR trick to put 0 into its argument.

```
150  def ZERO(a)
151    XOR a, a, a
152  end
```

In lines 153-166 there is an ADC() function implementing 16-bit summation with carry. In line 157 it calls FADD() sixteen times and then copies the result to r in line 158.

Please consider the lines 159-166. After iterating the FADD() function we have CARRY_REG equal to 0x0000 if there was no overflow during summation, or 0x0001 if there was. As you remember, where we were talking about the BRANCH() function, we said that it expects the condition to be either 0x0000 (false) or 0xFFFF (true). We are going to use CARRY_REG as the condition later on, so we have to convert 0x0001 to 0xFFFF.

In lines 159-165 we rotate the values of CARRY_REG (0x0001) across all sixteen bits of the word and in each position we OR it with t. By doing that we're spreading that 1 in the lowest significant position to all the bits in the word, and eventually 0x0001 becomes 0xFFFF.

```
153  def ADC(a, b, r)
154    mask, t = static :ADC_mask, :ADC_t
155    ZERO t
156    MOVi 0x0001, mask
157    16.times { FADD mask, CARRY_REG, a, b, t }
158    MOV t, r
159    ZERO t                      # Set t = 0.
160    16.times do                 # Rotate "CARRY_REG" 16 times.
161      OR t, CARRY_REG, t        # t |= CARRY_REG
162      MOV CARRY_REG, CARRY_REG  # SHIFT_REG = CARRY_REG << 1
163      MOV SHIFT_REG, CARRY_REG  # CARRY_REG = CARRY_REG << 1
164    end
165    MOV t, CARRY_REG            # CARRY_REG = t
166  end
```

In lines 167-175 there are two wrappers of ADC performing summation without using CARRY_REG.

```
167  def ADD(a, b, r)
168    ZERO CARRY_REG
169    ADC a, b, r
170  end
171  def ADDi(a, imm, r)
172    t = static :ADDi_t
173    MOVi imm, t
174    ADD a, t, r
175  end
```

In lines 176-187 there are functions for conditional branching. The main idea is to use OR() (line 182) as a decision maker.

```
176  # Jump 'a', if cond = FFFF, and 'b' if cond = 0000
177  def BRANCH(a, b, cond)
178    t1, t2 = static :BRANCH_t1, :BRANCH_t2
179    AND a, cond, t1     # t1 = a & cond
180    NOT cond, t2        # t2 = !cond
181    AND b, t2, t2       # t2 = b & t2 = b & !cond
182    OR t1, t2, IP       # ip = (a & cond) | (b & !cond)
183  end
184  # Jump 'a', if cond = FFFF, and 'b' if conf = 0000
185  def BRANCHi(a, b, cond)
186    BRANCH const(a), const(b), cond
187  end
```

In line 188-194 there is a function we haven't seen yet. It checks whether the a argument is zero or not. It adds 0xFFFF to a, and there is only one value of a possible when the overflow doesn't happen and CARRY_REG is 0 afterwards. This value is 0. If a is non-zero, there will be an overflow and CARRY_REG will become 0xFFFF. Then we negate the value of CARRY_REG and assign it to ZERO_REG.

So, if a is 0 this function sets ZERO_REG to 0xFFFF, or 0x0000 otherwise.

```
188  # if a != 0 then ZERO_REG = FFFF else ZERO_REG = 0000
189  def IS_0(a)
190    t = static :IS_0_t
191    ZERO CARRY_REG
192    ADC a, const(0xFFFF), t
193    NOT CARRY_REG, ZERO_REG
194  end
```

In lines 709-716 there are two functions doing conditional branching. They almost repeat the functionality of BRANCHi except that if the condition is false they carry on execution from the next instruction. This is implemented by introducing a bypass label pointing to an address right after the macro.

```
195  # ip = (ZERO_REG == FFFF ? a : ip)
196  def JZi(a)
197    bypass = make_label
198    BRANCHi a, bypass, ZERO_REG
199  label bypass
200  end
201  # ip = (ZERO_REG == FFFF ? a : ip)
202  def JNZi(a)
203    bypass = make_label
204    BRANCHi bypass, a, ZERO_REG
205  label bypass
206  end
```

In lines 725-224 there are functions doing different kinds of shifts. They all are based on the SHIFT_REG register.

```
207  def ROL(a, b)
208    MOV a, a              # SHIFT_REG = a << 1
209    MOV SHIFT_REG, b
210  end
211  def ROR(a, b)
212    t = static :ROR_t
213    MOV a, t
214    15.times { ROL t, t }
215    MOV t, b
216  end
217  def SHL(a, b)
218    ROL a, b
219    ANDi b, 0xFFFE, b
220  end
221  def SHR(a, b)
222    ROR a, b
223    ANDi b, 0x7FFF, b
224  end
```

The CRC-16 Algorithm

All right, at last in lines 225 to 251 we have our CRC-16 algorithm implemented via the functions we created previously. There are two loops. The outer one (crc_loop label) iterates over the source string via the ptr pointer up to the \x00 character. The inner loop iterates eight times over the current byte and applies the CRC-16 routine.

Ignoring for a moment what this code is doing, let's just reflect on it. It doesn't really look like Ruby. It looks like assembly language!

```
225  # NORCPU code
226  needle = literal Example + "\x00"
227  i, ch, t, ptr, crc16 = var 5
228  label :entry
229        MOVi 0xFFFF, crc16
230        MOVi needle, ptr
231  label :crc_loop                  # crc_loop:
232        PEEK ptr, ch               #   ch = *ptr
233        IS_0 ch                    #   is ch == 0?
234        JZi  :exit                 #   if yes, goto "exit"
235        ANDi ch, 0xFF, ch          #   ch &= 0xFF
236        XOR  crc16, ch, crc16      #   crc16 ^= ch
237        MOVi 8, i
238  label :loop_i                    # crc_loop_i:
239        ANDi crc16, 1, t           #   t = crc16 & 1
240        SHR crc16, crc16           #   crc16 >>= 1
241        IS_0 t                     #   is t == 0?
242        JZi :skip_xor              #   if yes, goto "skip_xor"
243        XORi crc16, 0x8401, crc16  #   crc16 ^= 0x8401
244  label :skip_xor                  # skip_xor:
245        ADDi i, 0xFFFF, i          #   i -= 1
246        IS_0 i                     #   is i == 0?
247        JNZi :loop_i               #   if not, goto "loop_i"
248        ADDi ptr, 0x0001, ptr      #   ptr += 1
249        JMPi :crc_loop             #   goto crc_loop
250  label :exit
251        EXITi()
```

In lines 252-278 we build our assembly. In lines 253-257 we join all the parts (code, labels, variables, constants, and strings) together into the assembly array. The labels have a # prefix.

```
252  assembly = []
253  Vars.each { |k, v| assembly.push "#%s" % k, v }
254  Text.each { |x| assembly.push x }
255  Statics.each { |k, v| assembly.push "#%s" % k, v }
256  Consts.each { |k, v| assembly.push "#%s" % v, k }
257  Literals.each do |k, v|
258    assembly.push "#%s" % k
259    v.each_byte { |x| assembly.push x }
260  end
```

In lines 261-270 we walk through the assembly, calculate the labels' addresses, and store them into the labels dictionary.

```
261  offset = 0
262  labels = {}
263  assembly.each do |x|
264    x = x.to_s
265    if x.start_with? '#' then
266      labels[x[1..-1]] = offset
267    else
268      offset = offset + 1
269    end
270  end
```

Finally, in lines 271-278 we cut out the labels from the assembly and replace the symbolic label names with concrete values.

```
271  # Remove all list having labels.
272  assembly.delete_if { |x| x.to_s.start_with? "#" }
273  # Substitute labels by values.
274  assembly.collect! do |x|
275    x = x.to_s.strip
276    subst = labels[x]
277    (if subst == nil then x else subst end).to_i
278  end
```

Now we have our code fully compiled and built into a sequence of NOR commands. It is in the assembly array.

In line 279-297 there is the NOR CPU emulator—the heart of our long journey. It executes the NOR code taken from the assembly until a halt condition when IP becomes equal to 0xFFFF (remember the EXITi macro?)

```
279  def nor(a, b)
280    ~(a | b) & 0xFFFF
281  end
282  ip = labels[IP]
283  shift_reg = labels[SHIFT_REG]
284  puts "Start"
285  mem = assembly
286  while true do
287    i = mem[ip]
288    a = mem[i + 0]
289    b = mem[i + 1]
290    r = mem[i + 2]
291    mem[ip] = i + 3
292    f = nor(mem[a], mem[b])
293    mem[r] = f
294    mem[shift_reg] = ((f >> 15) & 1) | ((f & 0x7FFF) << 1)
295    break if mem[ip] == 0xFFFF
296  end
```

Once the execution is complete, we print the content of the crc16 variable, which should contain our CRC16 value (line 297).

```
297  puts "NOR CPU CRC16: %04X" % mem[labels[crc16]]
```

In lines 298-310 we calculate CRC16 again using a purely Ruby implementation to make sure that our wonderful NOR-based program works correctly.

```
298  def calc_crc16(data)
299    crc = 0xFFFF
300    data.each_byte do |x|
301      crc = crc ^ (x & 0xFF)
302      8.times do
303        shift = (crc & 1) != 0
304        crc = crc >> 1
305        crc = crc ^ 0x8401 if shift
306      end
307    end
308    return crc
309  end
310  puts "Ruby CRC16: %04X" % calc_crc16(Example)
```

Wow! We're done!

At last we can run it (presuming you have the source in the file norcpu.rb).

```
ruby norcpu.rb
```

And hopefully it should print:

```
Start
NOR CPU CRC16: F6AD
Ruby CRC16: F6AD
```

If the values match, it proves that our NOR-based CPU works.

To play with the source, you can download it from the magazine website [U1].

Reflecting on what we have done, you might notice a serious drawback to our CPU: the number of NOR instructions to do even simple calculations is really big. For example, the size of our assembly code for calculating the tiny CRC16 algorithm is already 20273 words, a third of our memory size! So, an implementation of more sophisticated algorithms would require us to change the memory model to be at least 32-bit.

To be honest I don't see any real applications for such a CPU, but in educational terms I believe it has served well. Anyway, I hope it was a fun little exercise!



About the Author

Alexander Demin is a software engineer and Ph.D. in Computer Science. Constantly exploring new technologies, he believes that something amazing is always out there. He can be contacted at alexander@demin.ws [U2] or found at his home page [U3] or blog [U4].

Send the author your feedback [U5] or discuss the article in the magazine forum [U6].

External resources referenced in this article:

[U1] http://www.pragprog.com/magazines/2012-03/code/normachine/norcpu.rb

[U2] mailto:alexander@demin.ws

[U3] http://demin.ws

[U4] http://easy-coding.blogspot.com

[U5] mailto:michael@pragprog.com?subject= normachine

[U6] http://forums.pragprog.com/forums/134

Scala for the Intrigued

Pattern Matching

by Venkat Subramaniam

In this seventh installment of his series on Scala, Venkat shows how Scala does pattern matching.



You've seen quite a few examples of Scala's expressive and concise nature in this series. Pattern matching is another area where you'll appreciate the elegance of Scala. When programming you constantly make decisions based on the types of data and their values. Scala provides simple yet powerful facilities to program this common operation, as you'll see in this article.

To explore the expressive power of pattern matching, let's write a function that will count the number of nickels, dimes, and quarters in a given collection:

```
val coins = List(25, 10, 5, 10, 5, 25, 10, 5, 5, 25)
```

Using the traditional if-else construct, the code would look like this:

```
def countCoins(coins : Seq[Int], nickels : Int = 0,
  dimes : Int = 0, quarters : Int = 0)  : Map[String, Int] = {
  if (coins.isEmpty)
    Map("nickels" -> nickels, "dimes" -> dimes,
      "quarters" -> quarters)
  else {
    if (5 == coins.head) {
      countCoins(coins.tail, nickels + 1, dimes, quarters)
    }
    else {
      if (10 == coins.head) {
        countCoins(coins.tail, nickels, dimes + 1, quarters)
      } else {
        if (25 == coins.head) {
          countCoins(coins.tail, nickels, dimes, quarters + 1)
        } else {
          countCoins(coins.tail, nickels, dimes, quarters)
        }
      }
    }
  }
}

println(countCoins(coins))
//Map(nickels -> 4, dimes -> 3, quarters -> 3)
```

The function accepts a collection of coins as the first parameter, followed by the known number of nickels, dimes, and quarters, all set to 0 by default. Within the method, if there are no coins, you return a map of the known counts. Otherwise, you check the first coin using a series of simple if-else statements and recursively process the rest of the coins. You could have used the collection's filter method for this also, but in this example, we'll see how to use patterns instead.

That's quite a bit of code and it appears noisy. But we can use pattern matching to make this function concise. Scala provides a match method that appears

something like a switch statement, but is more powerful. It is an expression that can filter or match the target object against different types and data values. The previous code will reduce to a few lines that are easier to follow.

```
def countCoins(coins : Seq[Int], nickels : Int = 0,
  dimes : Int = 0, quarters : Int = 0)  : Map[String, Int] = {
  if (coins.isEmpty)
    Map("nickels" -> nickels, "dimes" -> dimes,
      "quarters" -> quarters)
  else coins.head match {
    case 5 => countCoins(coins.tail, nickels + 1,
      dimes, quarters)
    case 10 => countCoins(coins.tail, nickels,
      dimes + 1, quarters)
    case 25 => countCoins(coins.tail, nickels,
      dimes, quarters + 1)
    case _ => countCoins(coins.tail, nickels,
      dimes, quarters)
  }
}

println(countCoins(coins))
//Map(nickels -> 4, dimes -> 3, quarters -> 3)
```

Here you replaced the outermost else block with a call to the match method on the first element or the head of the coins collection. Scala matches the value of the first element with each case of literals you have provided and takes the appropriate route.

The underscore in case _ represents a wildcard, used if all of the other cases fail to match. Scala does not insist that you provide the wildcard case, but without it any mismatch would result in a runtime exception.

The match operation does not restrict you to simple literals. You can replace the top level if with a match in the coins count example:

```
coins match {
  case Seq() =>
    Map("nickels" -> nickels, "dimes" ->
      dimes, "quarters" -> quarters)
  case _ =>
    coins.head match {
      case 5 => countCoins(coins.tail, nickels + 1,
        dimes, quarters)
      case 10 => countCoins(coins.tail, nickels,
        dimes + 1, quarters)
      case 25 => countCoins(coins.tail, nickels,
        dimes, quarters + 1)
      case _ => countCoins(coins.tail, nickels, dimes, quarters)
  }
}
```

The first case takes effect if the collection is empty, and the wildcard case takes effect otherwise. You used two levels of match in this example, but that's not required. The list match also permits you to probe into the contents of the list. Using this you can let the top level match check the values of the list and eliminate the nested match.

```
  coins match {
    case Seq() =>
      Map("nickels" -> nickels, "dimes" ->
        dimes, "quarters" -> quarters)
    case Seq(5, _*) => countCoins(coins.tail, nickels + 1,
      dimes, quarters)
    case Seq(10, _*) => countCoins(coins.tail, nickels,
      dimes + 1, quarters)
    case Seq(25, restOfTheCoins @ _*) =>
      countCoins(restOfTheCoins, nickels, dimes, quarters + 1)
    case _ => countCoins(coins.tail, nickels, dimes, quarters)
  }
}
```

The pattern Seq(5, _*) matches a list whose first element is a value 5. The remaining elements in the list are represented by _* and may be captured into a reference, as in the case for 25. Capturing the values is optional, as you see in the case expressions.

But so far we've just scratched the surface of pattern matching. Scala allows you to match based on values and types; you can even use guarded cases using if, as you see next.

```
def process(message : Any) = {
  message match {
    case "hello" => println("received hello")
    case x : String => println("received a string " + x)
    case (a, b) => println("received tuple (" + a + ", " + b + ")")
    case 22 => println("received 22")
    case x : Int if x < 0 => println("received a negative number " + x)
    case y : Int => println("received number " + y)
  }
}

process(1, 2) // received typle (1, 2)
process("hello") // received hello
process("test")  // received a string test
process(22)      // received 22
process(12)      // received number 12
process(-32)     // received a negative number -32
```

The patterns "hello" and 22 look for an exact match. The patterns that specify the type, like x : String and y : Int, will accept any value of the appropriate type. The guarded case expects the value to satisfy the pattern in the case, type Int in this example, and the if statement to evaluate to true. You can also match tuples, which are lightweight immutable collections of values. In this example, you match any tuple with two values of any type. You can also dictate the type, like so: (a : Int, b : Double, c : String).

The order in which you place the case expressions matters. For example, if you reverse the last two cases, Scala will raise an unreachable code compile-time error. In a sense the pattern matching in Scala is like a coin sorter: the data falls into a matching slot as it flows over the cases.

You can match against various values and types, including your own types. Scala's special type of classes named *case classes* are very useful for this. The instances of case classes are immutable and they allow easy matching and extraction of data, as in the next example.

```
case class Sell(symbol : String, quantity : Int)
case class Buy(symbol : String, quantity : Int)

def processTrade(trade : Any) {
  trade match {
    case Sell(symbol, quantity) =>
      println("Selling " + quantity + " stocks of " + symbol)
    case Buy(symbol, quantity) if (quantity > 1000) =>
      println("Buying lots..." + quantity + " stocks of " + symbol)
    case Buy(symbol, quantity) =>
      println("Buying " + quantity + " stocks of " + symbol)
  }
}

processTrade(Sell("AAPL", 200)) // Selling 200 stocks of AAPL
processTrade(Buy("AAPL", 2000)) // Buying lots...2000 stocks of AAPL
processTrade(Buy("GOOG", 200))  // Buying 200 stocks of GOOG
```

When you create an instance of a case class, the values for the parameters flow in. For example, Sell("AAPL", 200) sends AAPL for the symbol and 200 for the quantity. When you use a case class in a pattern match, the values flow out. For example in case Sell(symbol, quantity), the values present in the instance of case class are extracted into these parameters. So, if the case class contained GOOG and 200 for these two fields, these values are extracted into the variables s and q, respectively, if the pattern is defined as case Sell(s, q). This direction in which the data moves is referred to as apply (invoking a function and sending data to it) and unapply (extracting the data from the object, moving the data in the opposite direction). The unapply operation is quite useful in pattern matching, as you'll see next.

If you want to perform more powerful extractions and apply a series of pattern matching, you can use Scala extractors. In the next example you'll write an extractor to parse the details of a task. First define a Task singleton class with an unapply method.

```
object Task {
  def unapply(taskInfo : String) = {
    val parts = taskInfo.split("---")
    if(parts.size != 2) None else Some(parts(0), parts(1))
  }
}
```

The unapply method expects the given string to contain one delimiter ---, and if the parameter conforms to that, returns as a tuple the two parts before and after this delimiter. If the format does not conform, it returns a None, indicating a failure of extraction. The unapply method can signal a failure by returning either a None or a false.

The values in the tuple returned from the unapply will be assigned to individual parameters you place in the case. You can use the Task as an extractor.

```
def processTask(detail : String) = {
  detail match {
    case Task(day, task) => println("Task for " + day + ":" + task)
    case _ => println("invalid task format")
  }
}
```

If the detail parameter conforms to the format that the Task extractor expects, the day and task parameters will contain the day and the task item for the day.

Exercise this function with a few calls:

```
processTask("Monday --- integrate with the payment system")
//Task for Monday : integrate with the payment system

processTask("Wednesday -- hack code with no tests")
//invalid task format

processTask("Frday --- discuss discount rates")
//Task for Frday : discuss discount rates
```

In the first call, the task conformed to the format and the extractor provided you the day and the task. In the second call, you ended up with a format error since the parameter did not conform to the expected format. The last call has the right format, but the day of week is messed up. This went undetected, but it's really easy to fix.

Create a DayOfWeek extractor to check the day of the week. The unapply method for this extractor can simply return a true or false to indicate if the string matches a day of the week.

```
object DayOfWeek {
  def unapply(day : String) = {
    List("Sunday", "Monday", "Tuesday", "Wednesday",
      "Thursday", "Friday", "Saturday") contains day.trim
  }
}
```

Let's improve the processTask method to use this extractor in combination with the extractor we already have.

```
def processTaskImproved(detail : String) = {
  detail match {
    case Task(day @ DayOfWeek(), task) =>
      println("Task for " + day + ":" + task)
    case _ => println("invalid task day or format")
  }
}
```

In the first case expression you first invoke the Task extractor. This parses the content and splits the day part from the task information part. You further invoke the DayOfWeek extractor on the day part. The value of day is stored in the day reference but the case match is not completed until the DayOfWeek extractor signals a success. If the day matches the format for a day of the week, as indicated by the unapply method of DayOfWeek, then the pattern match succeeds. If either the Task extractor signals a failure by returning None or the DayOfWeek extractor signals a failure by returning false, the match will not succeed. Exercise this improved version with the same arguments as you have sent to the first version of processTask and ensure that the validation of the data also happens.

```
processTaskImproved("Monday --- integrate with the payment system")
// Task for Monday : integrate with the payment system

processTaskImproved("Wednesday -- hack code with no tests")
// invalid task day or format

processTaskImproved("Frday --- discuss discount rates")
// invalid task day or format
```

In this article we continued our tour of Scala and looked at how Scala provides powerful pattern matching and extraction without compromising a bit on its expressive and concise nature. In the next part in this series, we'll explore a soft nature of Scala—the love it gives to something most of us have come to dread, XML processing.



About the Author

Dr. Venkat Subramaniam is an award-winning author, founder of Agile Developer, Inc., and an adjunct faculty at the University of Houston.

He has trained and mentored thousands of software developers in the US, Canada, Europe, and Asia, and is a regularly invited speaker at several international conferences. Venkat helps his clients effectively apply and succeed with agile practices on their software projects.

Venkat is the author of *.NET Gotchas,* the coauthor of 2007 Jolt Productivity Award winning Practices of an Agile Developer [U1], the author of Programming Groovy: Dynamic Productivity for the Java Developer [U2] and Programming Scala: Tackle Multi-Core Complexity on the Java Virtual Machine [U3]. His latest book is Programming Concurrency on the JVM: Mastering Synchronization, STM, and Actors [U4].

Send the author your feedback [U5] or discuss the article in the magazine forum [U6].

External resources referenced in this article:

[U1] http://pragprog.com/refer/pragpub33/titles/pad/practices-of-an-agile-developer

[U2] http://pragprog.com/refer/pragpub33/titles/vslg/programming-groovy

[U3] http://pragprog.com/refer/pragpub33/titles/vsscala/programming-scala

[U4] http://pragprog.com/refer/pragpub33/titles/vspcon/programming-concurrency-on-the-jvm

[U5] mailto:michael@pragprog.com?subject=scala

[U6] http://forums.pragprog.com/forums/134

Comparing Java and Scala's Expressiveness

A Case Study

by Brian Tarbox

Brian converts a Java program to Scala and ponders why the Scala version is so much smaller.



I recently rewrote my Open Source Log4JFugue music project [U1] from Java into Scala. As a relative newcomer to Scala I did this as an opportunity to immerse myself in a modest-sized project. When I was done I was surprised that the code size had shrunk from 2500 to 250 lines. In fact, given that I have about six months of Scala and about 12 years of Java experience, this result was shocking.

During the last month I've tried to understand what factors about Scala allow it to do so much in so few lines of code. The following factors seem to apply:

- richer library
- reduced ceremony
- reduced need to code for the lowest common denominator programmer
- transformational rather than procedural approach

Of these factors the transformational approach caught my attention. Object Oriented coding is supposed to be about telling objects to do their thing. However, Java code seems to have become big glops of procedural code where we describe *how* to do a thing. In short, it feels far more coupled and far less cohesive than we were promised Back In The Day.

Scala on the other hand feels more focused on the transformation of data. In part this is due to the extremely rich library of transformations provided for the various List classes.

As an experiment I decided to implement a log processing method in both Java and Scala and compare the results. The method provides a type of log analysis that we routinely performed in my Video On Demand group. As a measure of system health we monitored the time between successive log messages from particular packages. Short delays indicated code hot spots while very long delays indicated subsystems that might be stuck.

For this experiment we simplified the problem in several ways. We assumed that the log file could fit into memory and that all log messages take the form <long time value> <category> < message>, and we ignored exceptions.

The Java code is fairly straightforward, with the only tricky bit being lines 17 and 18, which take 144 characters to define two variables. The code creates a HashMap to hold the previous timestamp keyed by category. Each time a log message is processed, we scan it to find the time and category. We then get the previous time for that category from the HashMap and calculate the difference. We append that result to the category's vector of time differences in the TreeMap, taking care to initialize the vector the first time we see the

second time entry for each category. Lines 28 through 36 then print the results as a series of comma-separated lists.

```
 1  import java.io.BufferedReader;
 2  import java.io.DataInputStream;
 3  import java.io.FileInputStream;
 4  import java.io.InputStreamReader;
 5  import java.util.HashMap;
 6  import java.util.Iterator;
 7  import java.util.Map;
 8  import java.util.Scanner;
 9  import java.util.TreeMap;
10 import java.util.Vector;
11  void getTimeDiffGroupedByCat() {
12      FileInputStream fstream = new FileInputStream("textfile.txt");
13      DataInputStream in = new DataInputStream(fstream);
14      BufferedReader br = new BufferedReader(new InputStreamReader(in));
15      String strLine;
16      Long thisTime;
17      HashMap<String, Long> lastCatTime = new HashMap<String, Long>();
18      TreeMap<String, Vector<Long>> catTimeDiffs
          = new TreeMap<String, Vector<Long>>();
19      while ((strLine = br.readLine()) != null)   {
20              Scanner scanner = new Scanner(strLine);
21              thisTime = scanner.nextLong();
22              String category = scanner.next();
23              Long oldCatTime = lastCatTime.put(category, thisTime);
24              if(oldCatTime != null) {
25                    if(catTimeDiffs.get(category) == null) {
26                      catTimeDiffs.put(category, new Vector<Long>());
27                    }
28                    catTimeDiffs.get(category).add(thisTime - oldCatTime);
29              }
30      }
31      for(Map.Entry<String, Vector<Long>> thisEntry:
          catTimeDiffs.entrySet()) {
32              System.out.println("Category:" + thisEntry.getKey());
33              Iterator it = thisEntry.getValue().iterator();
34              while(it.hasNext()) {
35                      System.out.println(it.next());
36                      if(it.hasNext())
37                              System.out.print(", ");
38              }
39      }
40 }
```

Now here's the Scala version:

```
1   import Source.fromFile
2   def getTimeDiffGroupedByCat = {
3     val lines = fromFile("file.txt").getLines
4      val tuppleList = for(oneLine <- lines) yield {val z = oneLine.split
         (' '); (z(0).toInt, z(1)) }
5      for(groupedList <- tuppleList.toList.groupBy(oneTuple =>
         oneTuple._2)) {
6        val diffList = for(logPairs <- groupedList._2.sliding(2)) yield
           (logPairs(1)._1 - logPairs(0)._1)
7        println(groupedList._1 + ":" + diffList.mkString(","))
8    }
9   }
```

Considerably shorter. But what is that Scala version *doing*?

The Scala version takes a different approach, performing transforms on lists rather than manipulating mutable HashMaps. Since many readers will be unfamiliar with Scala, and since there are so few lines, we'll go through the code essentially line by line.

Line three creates a list of the lines in the file; we don't provide an explicit type for lines because the compiler infers that from the getLines function.

Line four iterates over the list of lines and for each one executes the anonymous code block after the yield. The result of this code block is added to the left-side variable tuppleList. While Java can only return a single object from a method, Scala supports the concept of a Tuple. The anonymous code block splits the log message line and then creates a two-element tuple of the first two entries of the split. So, line 3 transforms a List of Strings into a List of (timestamp, category) tuples. Not bad for a single line.

Line five almost looks like a database query in its use of the groupBy method, and the principle is the same. The groupBy method takes a list and creates a list of lists. Each top-level list consists of the value being grouped; i.e., the category and the sublist of all entries of the main list with that category value. In this case we group on oneTuple._2 which means the second item in the tuple.

Line six is called for each of the category sublists. It uses the sliding(2) method to pass a sliding window of size two over the list…in effect, looking at adjacent pairs of log messages of the same category. For each such pair it calculates the time difference between the first element of each tuple; i.e., logPairs(0)._1. Each such difference is added to the resulting diffList.

Line seven uses the underappreciated mkString method. This method converts a list into a string, with each element separated by the supplied string argument, in this case a comma. This single method call replaces lines 18 through 26 of the Java code and removes the worry of coding up a fencepost error in the list processing.

Several Observations

Starting with the import statements, Scala assumes that you might use the Scala library classes and for the most part imports them for you. If you are not familiar with Scala's collections library, the preceding code will have been be challenging. By the same token, if you are not familiar with Java Generics the following line is also challenging:

```
HashMap<String, Vector<Long>> catTimeDiffs = new HashMap<String,
  Vector<Long>>();
```

In the Java code one has to pick through the scaffolding and boilerplate to find the algorithmic code. In the Scala version all of the code contributes to the algorithm. If you prefer Java over Scala, you might say this shows that Scala is too terse. If you prefer Scala over Java, you might say this shows that Java is too verbose.

In the Java version, the main data structures, the HashMaps, are constantly modified. In the Scala version, all of the variables are in fact vals, which means they are immutable. While this program isn't multi-threaded, one could imagine

a version that was. In such a case the Java mutable HashMaps would need synchronization protection while the Scala immutable vals would not.

When I code in Scala I feel like I'm coding in a higher level language. In this example, the problem statement solved by the function is basically "show the differences between successive log messages grouped by category." The scala code is summarized by "for the lines in the file build a set of lists grouped by category and compute their differences." Those summaries sound remarkably similar. The Scala code seems like a description of what to do, while the Java code is a description of how to do it ("add an element to the Vector inside the TreeMap unless that Vector is null in which case insert a new one….")

Another way of looking at it is this. How many times in your career have you written the code to print a comma-separated list? How many times did you just ignore the extra comma after the last item in the list? How many times did you get the code exactly right the first time? Would you like all that accumulated time back? With the mkString function, no programmer should ever have to reinvent this particular wheel again. In Java before you can hammer a nail you have to go build a hammer. Scala just gives you the hammer and it's up to you not to hit your thumb with it!



About the Author

Brian Tarbox is a Distinguished Member of Technical Staff at Motorola in the Systems Engineering group designing next generation video products. He writes a blog on the intersection of software design, cognition, music, and creativity at briantarbox.blogspot.com [U2]. His primary contact point is about.me/BrianTarbox [U3].

Send the author your feedback [U4] or discuss the article in the magazine forum [U5].

External resources referenced in this article:

[U1] http://pragprog.com/magazines/2009-09/and-your-bugs-can-sing

[U2] http://briantarbox.blogspot.com/

[U3] http://about.me/BrianTarbox

[U4] mailto:michael@pragprog.com?subject= scala

[U5] http://forums.pragprog.com/forums/134

Four Common Pairing Pitfalls and How to Avoid Them

Four Antipatterns of Pair Programming

by Zee Spencer

Pairing is hard. Here are some tips to make it easier.



Pairing is hard; really hard. Not just because pairing can literally stink. Mints and honest, compassionate feedback take care of that. Not because it's technically difficult. Anyone can sit next to someone else for a few hours, point out typos and feel good about "contributing value."

No, pairing is inherently hard. Think about it. It forces you to connect with someone for an extended period of time. And that person may not share your background, skills, passion, or even vocabulary. Yet when pairing you must work together to accomplish a shared goal.

I've thought a lot about exactly what makes pairing hard. While actually doing pair programming, I've noticed some antipatterns that make pairing especially hard:

- Motive Mismatches
- Competence Chasms
- Disposition Dissonance
- Semantic Squabbles

Each of these antipatterns presents its own challenges. Each, though, offers potential benefits if you deal with it effectively. None of them can be solved by avoidance.

Motive Mismatches

There are many goals people have when pairing: momentum, focus, knowledge transfer, social connection, or fun. Normally these motives are mixed, with a few being most important in any given session. When goals are shared, pairing is fun and constructive. When they are mismatched, pairing can be unpleasant.

Things to Watch For:

A motive mismatch is revealed by a lack of momentum. You may stop and start abruptly, struggle to agree on what to do next, or become disengaged.

Where to Start:

Stop and have a short conversation with your pair. Directly and politely point out that something feels wrong. Explain your goals, ask for theirs, and create shared goals. When goals are significantly different, acknowledge this and reassess them regularly so that both partners get to meet their expectations.

Competence Chasms

We all have things we're good at. This is a good thing: diversity of thought, skills, and experiences can lead to creative yet simple solutions. Novices often

spot problems by asking questions that jiggle their partner's mental state enough to provide a new solution.

But a competence chasm can lead to chaos. Trying to explain the concept of constraints and how proximity implies relationship when your partner lacks a design background can be frustrating when all you you want to do is solve the issue and move on.

Things to Watch For:

You may feel tired or frustrated, seem to only have arguments, or a partner who is afraid to say anything.

Where to Start:

When the goal is to share knowledge it may be all right that you are making little progress on the task at hand. When you get tired, take a break. This relieves the exhaustion that can occur when pairing with someone at a different skill level.

In any case, when disagreeing you may want to defer to your pair and ask for further reading or offer to discuss the idea over lunch. Regardless of experience, your pair offers insights that would be foolish to ignore just because they don't know how to program, test, or design as well as you.

Disposition Dissonance

We are all different people. We have different personalities, different passions, and different behavioral triggers. These differences can make pairing hard. Someone who doesn't talk much can feel oppressed by a partner who laughs and jokes all day long. Someone who struggles with self esteem could be devastated by a harsh word casually spoken.

Things to Watch For:

Watch your pair partner. Gauge their emotional state. If they are frustrated or disengaged or avoid pairing with you in the future it can be because of personality differences.

Where to Start:

First, remember that your pair partner is a person. Sometimes their behavior seems odd, unsettling, or disturbing, but it is never wrong. Once you accept the fact that the person you're working with is behaving the way they behave because that is how they are, it becomes much easier to stop, reflect, and determine a valid course of action.

You may find it useful to look into personality frameworks such as DISC [U1] system. Disc is a competitor to the Myers Briggs [U2] personality profile system. I personally prefer DISC because it chooses not to declare what someone is, and instead focuses on what their behavioral tendencies are. By understanding the strengths and weaknesses of your own and other personality types, you can gain insight into what you are doing that makes your pair upset.

Semantic Squabbles

Different backgrounds result in different phrasing. What one calls a "resource" another may demand be called a "person." There is no malice on either's part,

but such semantic squabbles can result in hurt feelings and damaged relationships.

Things to Watch For:

Words are powerful, and often create powerful emotions. Check your emotional state if you hear a trigger word and be aware of your pair's triggers.

Where to Start:

Your pair calling your project backlog and burnup a forecast does not necessarily mean they are treating it like a traditional project plan.

We all have words that trigger us. When you are triggered, take a few breaths, ask a few questions, and try to understand the other's perspective. When your pair is triggered, take a step back. Apologize, and replace your word with theirs.

Yes, pairing is hard. But if you watch for these antipatterns and address them when you spot them, the rewards more than compensate you for your effort.



About the Author

Zee loves to make things and teach people. He blends a focus on experience design, technical practice, and lean project management into a version of software akido. This allows him to help teams learn to consistently deliver more of what matters. When not making software Zee jams with friends, rock climbs, and watches an absurd amount of science fiction. You may stalk him on the internet at @zspencer [U3].

Send the author your feedback [U4] or discuss the article in the magazine forum [U5].

External resources referenced in this article:

[U1] http://en.wikipedia.org/wiki/DISC_assessment

[U2] http://en.wikipedia.org/wiki/Myers-Briggs_Type_Indicator

[U3] http://www.twitter.com/zspencer

[U4] mailto:michael@pragprog.com?subject= pairing

[U5] http://forums.pragprog.com/forums/134

Calendar

Author sightings, partner events, and other notable happenings.



Here's what's coming up in the next three months:

Author Appearances

Who's where, and what for.

March 6-9 **iPhone/iPad Programming**
Daniel H Steinberg, Dim Sum Thinking [U1]
Pragmatic Studio, [U2] Denver, CO

Mar 8 **How to approach Refactoring?**
Venkat Subramaniam, the author of *.NET Gotchas,* the coauthor of 2007 Jolt Productivity Award winning Practices of an Agile Developer [U3], the author of Programming Groovy, [U4] Programming Scala [U5] and Programming Concurrency on the JVM [U6]
Scandinavian Agile Conference [U7], Helsinki, Finland

Mar 9–11 **Multiple talks on JavaScript, HTML 5, Scala, Concurrency, and automated testing**
Venkat Subramaniam, the author of *.NET Gotchas,* the coauthor of 2007 Jolt Productivity Award winning Practices of an Agile Developer [U8], the author of Programming Groovy, [U9] Programming Scala [U10] and Programming Concurrency on the JVM [U11]
New England Software Symposium [U12], Boston, MA

Mar 12-14 **Tutorial: "Influence and Authority: Using Your Personal Power to Get Things Done," Keynote: "QA or Test? Does it Matter? You Bet it Does!"**
Johanna Rothman, author of Manage Your Project Portfolio [U13]: Increase Your Capacity and Finish More Projects, Manage It! [U14]: Your Guide to Modern, Pragmatic Project Management, and Behind Closed Doors [U15]: Secrets of Great Management
Belgium Testing Days [U16], Brussels, Belgium

Mar 15 **TDB**
Venkat Subramaniam, the author of *.NET Gotchas,* the coauthor of 2007 Jolt Productivity Award winning Practices of an Agile Developer [U17], the author of Programming Groovy, [U18] Programming Scala [U19] and Programming Concurrency on the JVM [U20]
Java User Group [U21], NY, NY

Mar 15 **"A3 reports—processing change proposals"**
Staffan Nöteberg, author of Pomodoro Technique Illustrated [U22]
Devlin 2012 [U23], Linköping, Sweden

Mar 15—17 **Mountain West Ruby Conference Workshop: "Advanced Rails Design: Use Fancy OO Techniques Without Turning Ruby into Java"**
Noel Rappin, author of Rails Test Prescriptions: [U24] Keeping Your Application Healthy
Mountain West Ruby Conference Workshop [U25], Salt Lake City, UT

Mar 16 **Keynote**
Jonathan Rasmusson, author of The Agile Samurai [U26]
Agile Japan 2012 [U27], Osaka, Japan

March 16–17 **"Core Audio Cranks It Up" and "Core What?"**
Chris Adamson, co-author of iOS SDK Development [U28]
CocoaConf [U29], Chicago, IL

March 16-17 **Storyboarding, Testing, and a Keynote**
Daniel H Steinberg, Dim Sum Thinking [U30]
CocoaConf, [U31] Chicago, IL

Mar 19–20 **"Advanced Regex"**
Staffan Nöteberg, author of Pomodoro Technique Illustrated [U32]
Software Passion Summit 2012 [U33], Gothenburg, Sweden

Mar 19–21 **Keynote and talks on Concurrency, Scala, and JVM languages integration**
Venkat Subramaniam, the author of *.NET Gotchas,* the coauthor of 2007 Jolt Productivity Award winning Practices of an Agile Developer [U34], the author of Programming Groovy, [U35] Programming Scala [U36] and Programming Concurrency on the JVM [U37]
33rd Degree Conference [U38], Krakow, Poland

Mar 23–24 **"Don't Fear the Threads: Simply Your Life with JRuby"**
David Copeland, author of Build Awesome Command-Line Applications in Ruby [U39]
Ruby Nation 2012 [U40], Reston, VA

Mar 27 **"Don't Look Now, But You Might Be Agile"**
Ian Dees, author of Scripted GUI Testing With Ruby [U41] and co-author of Using JRuby [U42]
Embedded Systems Conference [U43], San Jose, CA

Apr 10—12 **"SQL Injection Myths and Fallacies" and "Full Text Search Throwdown"**
Bill Karwin, author of SQL Antipatterns: [U44] Avoiding the Pitfalls of Database Programming
Percona Live MySQL Conference & Expo 2012 [U45], Santa Clara, CA

Apr 10—12 **"The CoffeeScript Edge"**
Trevor Burnham, author of CoffeeScript: Accelerated JavaScript Development [U46]
Emerging Technologies for the Enterprise [U47], Philadelphia, PA

Apr 17—18 **Public Workshop: "Working Effectively In Geographically Distributed Agile Project Teams"**
Johanna Rothman, author of Manage Your Project Portfolio [U48]: Increase Your Capacity and Finish More Projects, Manage It! [U49]: Your Guide to Modern, Pragmatic Project Management, and Behind Closed Doors [U50]: Secrets of Great Management
Working Effectively In Geographically Distributed Agile Project Teams [U51], Pleasanton, CA

Apr 17—20 **Keynote and talks on multiple topics**
Venkat Subramaniam, the author of *.NET Gotchas,* the coauthor of 2007 Jolt Productivity Award winning Practices of an Agile Developer [U52], the author of Programming Groovy, [U53] Programming Scala [U54] and Programming Concurrency on the JVM [U55]
Great Indian Developer Summit [U56], Bangalore, India

Apr 19–20 **"Down the NoSQL Rabbit Hole"**
Eric Redmond, author of Seven Databases in Seven Weeks [U57]
Railsberry [U58], Krakow, Poland

Apr 21–22 **"Regex in JavaScript and HTML5"**
Staffan Nöteberg, author of Pomodoro Technique Illustrated [U59]
Dynamic Languages Conference 2012 [U60], Stockholm, Sweden

Apr 23 **"Mongo Indexing"**
Eric Redmond, author of Seven Databases in Seven Weeks [U61]
Mongo Meetup [U62], Boston, MA

Apr 23–24 **"Inclusion over Integration"**
Staffan Nöteberg, author of Pomodoro Technique Illustrated [U63]
Agile Sweden 2012 [U64], Stockholm, Sweden

Apr 27–29 **Open Space Event**
Venkat Subramaniam, the author of *.NET Gotchas,* the coauthor of 2007 Jolt Productivity Award winning Practices of an Agile Developer [U65], the author of Programming Groovy, [U66] Programming Scala [U67] and Programming Concurrency on the JVM [U68]
Simple Design and Testing Conference (SDTConf) [U69], Houston, TX

April 27—29 **Multiple talks on JavaScript, HTML 5, Scala, Concurrency, and automated testing**
Venkat Subramaniam, the author of *.NET Gotchas,* the coauthor of 2007 Jolt Productivity Award winning Practices of an Agile Developer [U70], the author of Programming Groovy, [U71] Programming Scala [U72] and Programming Concurrency on the JVM [U73]
Northern Virginia Software Symposium [U74], Reston, VA

May 4–5 **Multiple talks on JavaScript, HTML 5, Scala, Concurrency, and automated testing**
Venkat Subramaniam, the author of *.NET Gotchas,* the coauthor of 2007 Jolt Productivity Award winning Practices of an Agile Developer [U75], the author of Programming Groovy, [U76] Programming Scala [U77] and Programming Concurrency on the JVM [U78]
New York Software Symposium [U79], NY, NY

May 8–9 **"Project Portfolio Management"**
Johanna Rothman, author of Manage Your Project Portfolio [U80]: Increase Your Capacity and Finish More Projects, Manage It! [U81]: Your Guide to Modern, Pragmatic Project Management, and Behind Closed Doors [U82]: Secrets of Great Management
Sao Paulo, Brazil

May 8–12 **"Pomodoro–can you focus for 25 minutes?"**
Staffan Nöteberg, author of Pomodoro Technique Illustrated [U83]
Agile Africa 2012 [U84], Johannesburg, South Africa

May 18–20 **Multiple talks on JavaScript, HTML 5, Scala, Concurrency, and automated testing**
Venkat Subramaniam, the author of *.NET Gotchas,* the coauthor of 2007 Jolt Productivity Award winning Practices of an Agile Developer [U85], the author of Programming Groovy, [U86] Programming Scala [U87] and Programming Concurrency on the JVM [U88]
Lone Star Software Symposium [U89], Dallas, TX

May 19–24 **"Problem Solving Leadership"**
Johanna Rothman, author of Manage Your Project Portfolio [U90]: Increase Your Capacity and Finish More Projects, Manage It! [U91]: Your Guide to Modern, Pragmatic Project Management, and Behind Closed Doors [U92]: Secrets of Great Management
Problem Solving Leadership [U93], Albuquerque, NM

May 23–24 **"Advanced Regex"**
Staffan Nöteberg, author of Pomodoro Technique Illustrated [U94]
DevSum 2012 [U95], Stockholm, Sweden

May 28–29 **"Transitioning to Agile for Test (Testers, Test Managers, Project Managers and others who are not developers"**
Johanna Rothman, author of Manage Your Project Portfolio [U96]: Increase Your Capacity and Finish More Projects, Manage It! [U97]: Your Guide to Modern, Pragmatic Project Management, and Behind Closed Doors [U98]: Secrets of Great Management
Transitioning to Agile for Test [U99], Stockholm, Sweden

May 30 **"Project Portfolio Management"**
Johanna Rothman, author of Manage Your Project Portfolio [U100]: Increase Your Capacity and Finish More Projects, Manage It! [U101]: Your Guide to Modern, Pragmatic Project Management, and Behind Closed Doors [U102]: Secrets of Great Management
Project Portfolio Management [U103], Stockholm, Sweden

O'Reilly Events

Upcoming events from our friends at O'Reilly.

Mar 18 **Tools of Change Bologna: "TOC and the Bologna Children's Book Fair will team up once again, this time producing an unparalleled event featuring sessions, demos, workshops, and keynotes covering the art, craft, and business of storytelling in the digital age."**
TOC Bologna [U104], Bologna, Italy

Apr 2—4 **Where Conference: "The O'Reilly Where Conference is where the people working on and using location technologies come together to explore best practices and emerging trends in software development, tools, business strategies, and marketing."**
whereconf [U105], San Francisco, CA

May 29–31 **Fluent Conference: "The O'Reilly Fluent Conference is about everything JavaScript. If you're developing for the Web, desktop, or mobile, knowing the ins and outs of JavaScript and related technologies is critical. Come to Fluent to learn from expert developers who are using JavaScript in all kinds of contexts, to do things that no one ever expected JavaScript could do, and do so well."**
Fluent Conference, [U106] San Francisco, CA

USENIX Events

What's coming from our USENIX friends.

Apr 10—13 **EuroSys 2012: "EuroSys has become a premier forum for discussing various issues of systems software research and development, including implications related to hardware and applications."**
EuroSys 2012 [U107], Bern, Switzerland

Apr 24 **2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services: "The second Hot-ICE workshop builds on the success of the first workshop last year and seeks to bring together researchers and practitioners working on network and service management in the Internet, cloud, and enterprise domains."**
Hot-ICE '12 [U108], San Jose, CA

Apr 24 **5th USENIX Workshop on Large-Scale Exploits and Emergent Threats: "Now in its fifth year, LEET continues to be a unique forum for the discussion of threats to the confidentiality of our data, the integrity of digital transactions, and the dependability of the technologies we increasingly rely upon."**
LEET '12 [U109], San Jose, CA

Apr 25—27 **9th USENIX Symposium on Networked Systems Design and Implementation: "NSDI '12 focuses on the design principles, implementation, and practical evaluation of large-scale networked and distributed systems."**
NSDI '12 [U110], San Jose, CA

May 11–12 **Professional IT Community Conference: "The third annual Professional IT Community Conference (PICC) is requesting individuals who are interested in presenting papers, talks, or participating in panels on any aspect of IT and System Administration."**
PICC '12, [U111] New Brunswick, NJ

Other Happenings

Mar 1 **Yahoo (as a corporation) is 17.**

Mar 3 **GNOME first came out of its burrow on this date in 1999.**

Mar 9 **Jef Raskin would have turned 69 today.**

Mar 11 **J.C.R. Licklider would have turned 97 today.**

Mar 13 **On this date in 1986, Microsoft went public, creating four billionaires.**

Mar 15 **A-life pioneer Craig Reynolds, who worked on the first *Tron*, is 59.**

Mar 15 **It's the Ides of March. Beware.**

Mar 16 **Richard Stallman is 59 and MINIX author Andy Tanenbaum is 68.**

Mar 24 **It's Ada Lovelace Day, honoring women in technology and science.**

Mar 24 **Mac OS X is XI today.**

Mar 25 **On this day in 1995, Wiki went worldwide as Ward Cunningham installed WikiWikiWeb on the Web.**

Mar 26 **Larry Page is 39.**

Mar 31 **Evan Williams, co-founder of Twitter, is 40.**

Apr 6 **MacOS co-creator Andy Hertzfeld is 59.**

Apr 5 **Inventor Dean Kamen is 61.**

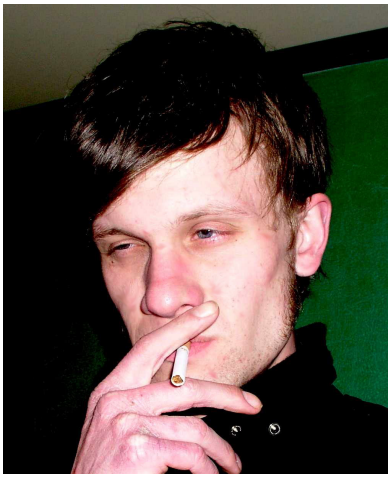
Apr 9	**ENIAC co-inventor J. Presper Eckert was born on this date in 1919.**
Apr 9	**SQLite architect D. Richard Hipp is 51.**
Apr 14	**Ruby designer Yukihiro Matsumoto (Matz) is 47.**
Apr 20	**Chief Yahoo David Filo is 46.**
Apr 24	**Champion of modelessness Larry Tesler is 67.**
Apr 28	**Mathematician Kurt Gödel was born on this date in 1906.**
Apr 30	**Information theory pioneer Claude Shannon was born on this date in 1916.**
Apr 30	**Colossal Cave Adventure creator Don Woods is 58.**
Apr 30	**The World Wide Web is 19.**
May	**In May, 1980, Erno Rubik's puzzle started selling internationally under the name "Rubik's Cube."**
May 1	**2006 MacArthur Fellow Jim Fruchterman is 53.**
May 2	**Protoblogger Dave Winer is 57.**
May 3	**Simon Tatham, known for playing with PuTTY, is 35.**
May 8	**jQuery creator John Resig got to age 28 one day ahead of Mark Zuckerberg.**
May 9	**Red Hat founder Marc Ewing is 44.**
May 11	**Edsger Dijkstra's 82nd birthday.**
May 14	**Zillionaire entrepreneur Mark Zuckerberg is 28.**
May 17	**Youthful Alan Kay is 72.**
May 19	**Today is the eleventh anniversary of the opening of the first Apple retail stores. Also, Java daddy James Gosling is 57 today; Theo de Raadt, founding member of NetBSD and founder of OpenBSD and OpenSSH, is 44; SNOBOL creator Ralph Griswold would have turned 78 today; and CP/M inventor Gary Kildall would have turned 70.**
May 26	**Ward Cunningham is 63.**
May 27	**PEZ was trademarked on this date in 1952.**
May 28	**Networking hotshot Adam Dunkels is 34.**
May 29	**63rd birthday of shareware co-creator Bob Wallace.**
May 31	**John Kemeny was born on this day in 1926.**

Shady Illuminations

Time Bandits

by John Shade

John is too laid back to spring forward.



A sick gag that came up last summer is repeating on us. I refer to the "brogrammer" meme, the sophomoric conceit that there are actually programmers who act like fratboys, belong to rowing teams, listen to the Dave Matthews Band, and call everyone "Bro." The Revenge of the Winklevosses.

I call Fail on this meme-manqué.

No good can come of it when people with no formal training in neologistics invent their own words. Because words breed. Before we're done with Natural Light-drinking brogrammers we'll be neck-deep in spurious fauxgrammers, dilatory slowgrammers, contrary nogrammers, non-pro amateurgrammers, and pompous and pedantic kelseygrammers.

(Hmm. Kelsey Grammer's prep school had a rowing team and he's the same age as Bill Gates. Could he have been a proto-brogrammer? Nah, I can't picture him hacking 8080 code for the Altair in his dorm room at the Julliard.)

But the real problem with the brogrammer gag is that it isn't plausible. We have too much solid scientific research on what programmers are really like.

Science tells us that good programmers are lazy, impatient, and filled with hubris [U1]. They are angry, pessimistic, cautious nerds. [U2] Also calm, adaptible, humble, assertive, and self-deprecating [U3]. These are uncontrovertible facts, the result of solid, objective research. I'd merely add that programmers are the kind of people who overthink it [U4]. A brogrammer might comment on a Rick Moranis movie, but the comment wouldn't involve the Schwartzchild radius.

In other words, there are no brogrammers, and this talk about them is taking up bits that could be dedicated to overthinking one of the real issues facing us today. Like the hour that's being cruelly stolen from many of us this month.

Sprung Logic

If you do a search on "converting 32-bit applications into 64-bit applications," you'll find a lot of complicated advice. It seems to me there's a really simple solution that's being overlooked. Just define 32 to be 64:

```
int 32 = 64;
```

or do this:

```
32 += 32;
```

Isn't that the logic behind Daylight Savings Time?

Daylight Savings Time [U5], or DST, is the legally-prescribed resetting of clocks twice a year: forward in the spring and back in the fall. Most of the world is

spared this nonsense, but we the the USA are afflicted by it, and this is the spring forward month.

As you can tell, I think DST is stupid. It messes with our circadian rhythms [U6] and puts us through a pointless routine twice a year of remembering to adjust all our clocks, remembering what direction to adjust them, remembering where all our clocks are (hint: they're not all in the house), and fixing the blame when our household clocks don't get adjusted correctly. DST causes accidents, reduces worker productivity, causes people to miss flights [U7], increases pollution [U8], and gives people heart attacks [U9]. And makes programmers write hideous code. Brogrammer code.

So what was the idea behind this changing of the clocks? That's a little unclear, at least to me. Apparently it involved entomology in New Zealand [U10]. And since the seasons are reversed in their hemisphere—no, I'm getting confused.

Or was the idea to save energy? If so, does it work? If it worked once, is that any reason to think that what worked for 1940s conditions still works today? The evidence is contradictory on that, but seems to say that the potential energy saving or cost of DST depends on things like the annual usage of air conditioning. So Indiana might be getting penalized in order to benefit Florida.

Or was if for the benefit of farmers? You hear that claim, but it appears to have no foundation [U11]. Dairy cows, in fact, are particularly vocal opponents of DST, objecting loudly to having their biological clocks messed with twice a year. Or so I hear.

Whatever. I have thought the matter over thoroughly and I say Daylight Davings Time is the bunk. Here's my alternative plan:

If you need to get up an hour earlier, get up an hour earlier.

It's a crazy idea, but it just might work.

About the Author

John Shade was born under a cloud in Montreux, Switzerland, in 1962. Subsequent internment in a series of obscure institutions of ostensibly higher learning did nothing to brighten his outlook. He says he's bitter over this Daylight Savings Time business and wants his hour back to fritter away unproductively, as is his wont. Send the author your feedback [U12] or discuss the article in the magazine forum [U13].

External resources referenced in this article:

[U1] http://analytics.ncsu.edu/sesug/2005/SER06_05.PDF
[U2] http://www.softwarebyrob.com/2006/08/20/personality-traits-of-the-best-software-developers/
[U3] http://www.dorsethouse.com/features/excerpts/expsych8.html
[U4] http://www.overthinkingit.com/2010/12/02/honey-i-shrunk-the-kids/
[U5] http://aa.usno.navy.mil/faq/docs/daylight_time.php
[U6] http://www.bluepoppy.com/blog/blogs/blog1.php/stupid-is-as-stupid-does-daylight-saving
[U7] http://midimagic.sgc-hosting.com/daystupid.htm
[U8] http://www.9news.com/news/local/article/253708/222/How-much-does-daylight-saving-time-cost
[U9] http://thinkprogress.org/romm/2010/03/13/205642/daylight-saving-time-energy-dst/
[U10] http://en.wikipedia.org/wiki/George_Vernon_Hudson
[U11] http://skeptoid.com/episodes/4172
[U12] mailto:michael@pragprog.com?subject=shade
[U13] http://forums.pragprog.com/forums/134

But Wait, There's More...

Coming Attractions

This may be the last page, but that doesn't mean it's the end of the conversation. Keep up with what's going on in the Pragmatic universe by subscribing to our newsletter and drop by the Pub again next month for more Pragmatism. We'll keep a seat warm and a mug chilled and a candle burning in the window.



Coming Soon in PragPub



That interview with Antonio Cangiano Technical Blogging[U1]) should happen next month.

Coming Soon on the Bookshelf



- Deploying with JRuby in beta
- Build Awesome Command-Line Applications in Ruby in print
- The dRuby Book: Distributed and Parallel Computing with Ruby in print
- The Rails View in print
- Rails Recipes: Rails 3 Edition in print

Right. But to really be in the know, you need to subscribe to our weekly newsletter. It'll keep you in the loop, it's a fun read, and it's free. All you need to do is create an account on pragprog.com (email address and password is all it takes) and select the checkbox to receive newsletters.



In the Meantime

While you're waiting for the next issue of the newsletter or of *PragPub* to come out, you can follow us on twitter at @pragpub, @PragmaticAndy, or @pragdave. Or on Facebook at facebook.com/PragmaticBookshelf[U2]. Or email us at michael@pragprog.com[U3].