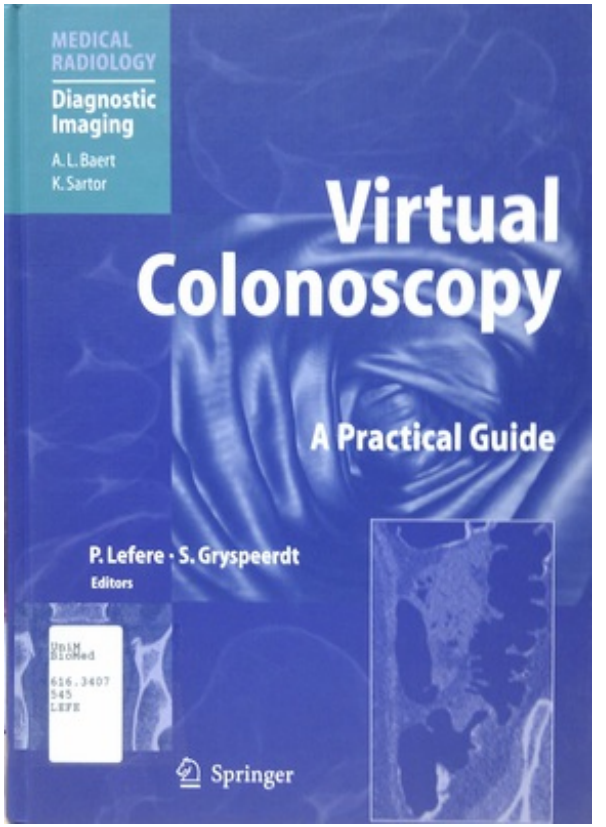
The First Law of Language Design: A Colonoscopy

Brad Bowman <first-law@bereft.net>
2010-11-25

The First Law of Language Design: A Colonoscopy



The First Law of Language Design:

The First Law of Language Design:

> "Everyone wants the colon"

Synopsis 01
— Larry Wall

The Second Law of Language Design:

> "Larry gets the colon for whatever he wants"

Synopsis 01
— Larry Wall

Colonoscopy

(noun) 1

visual examination of the colon (with a colonoscope) from the cecum to the rectum; requires sedation

(noun) 2

visual examination of the colon *:* (with a grep) from the C to the Ruby; induces sedation

Figure 1.15 The first functional fibersigmoidoscope—four-way angling: Eder Instrument Co. (Overholt, 1963).

Colonoscopy 2 (noun)

- How does the First Law fit other programming languages?
- Does the use of the colon tell us something about a language?
- Can it be used as a quick evaluation of languages?

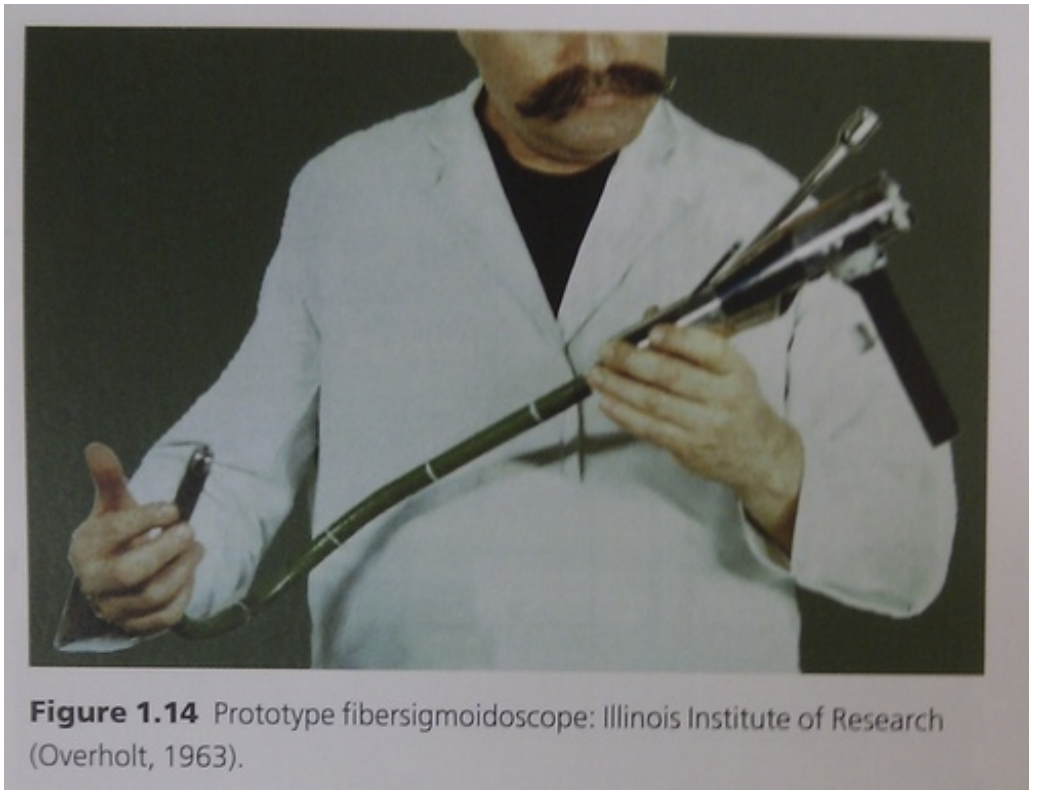
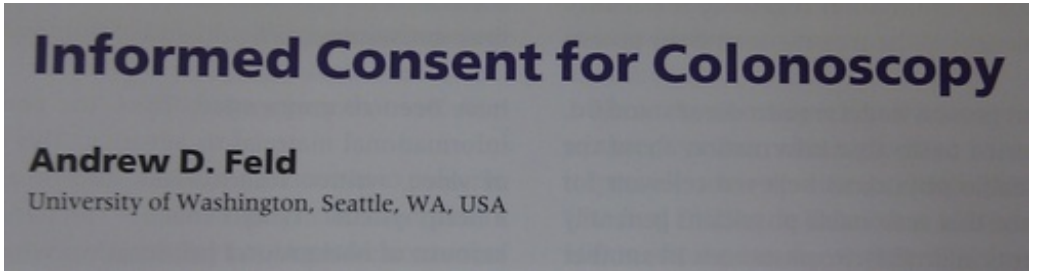


Figure 1.14 Prototype fibersigmoidoscope: Illinois Institute of Research (Overholt, 1963).

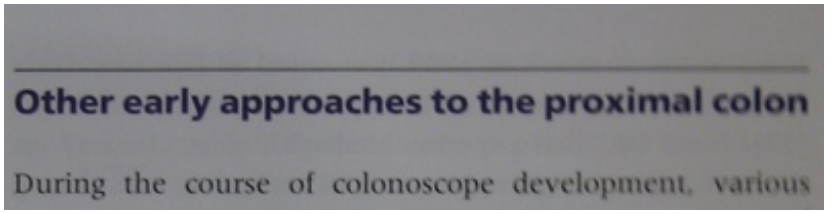
The Fine Print - IANALL

I am not a language lawyer. This was cobbled together from a pile of quickrefs, Wikipedia and Rosetta I don’t know all these languages. Many of them I don’t want to know. So please, let’s try to get through this quickly. It’ll hurt less that way. Sorry if I haven’t covered your favourite language, consider it homework. Emphasis on the "home". Only DEFCON 1 interruptions please, eg. "You’re on fire". The categories and paradigms are only meant to be "close enough", most languages cross-over, at least to a degree. This isn’t science. This isn’t even "vial of green, glowing stuff science". This is a joke gone too far. The slides will probably be on github, so fork off and fix the bugs.



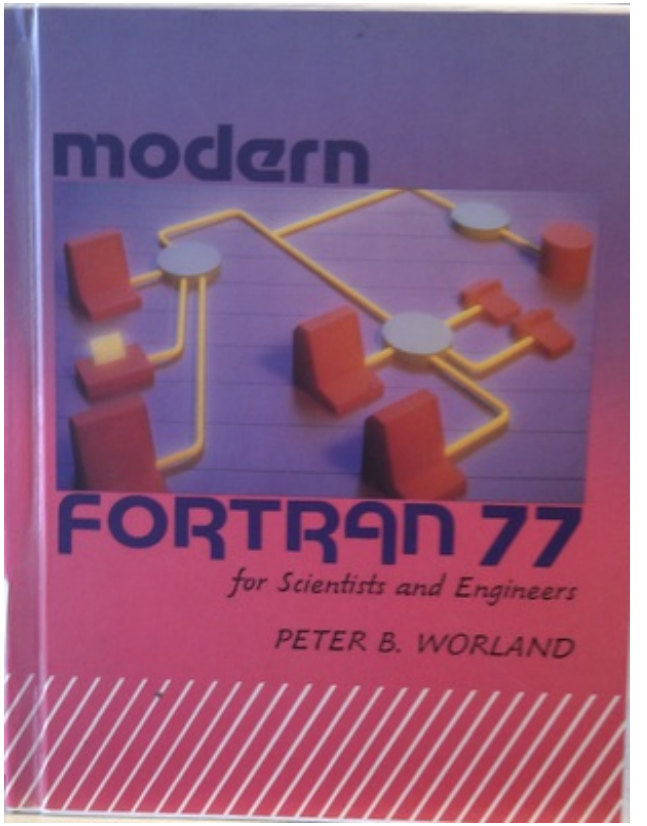
Old-school Languages

- Fortran
- APL
- Forth
- C
- (Lisp later)



Fortran

- Scientific Numeric Programming
- Fast for High-end Simulations
- Crunching Arrays of Floats
- Dated - GOTO, Size Limits, CAPITALS, punch-cardy



Fortran Colon

Range in SELECT/CASE

```
SELECT CASE (cmdchar)
  CASE ('1':'9')
    CALL RetrieveNumFiles (cmdchar)
```

Type tags, later Fortran versions

real, intent(in), dimension(:)

b, A(:,:)

Fortran Colon 2

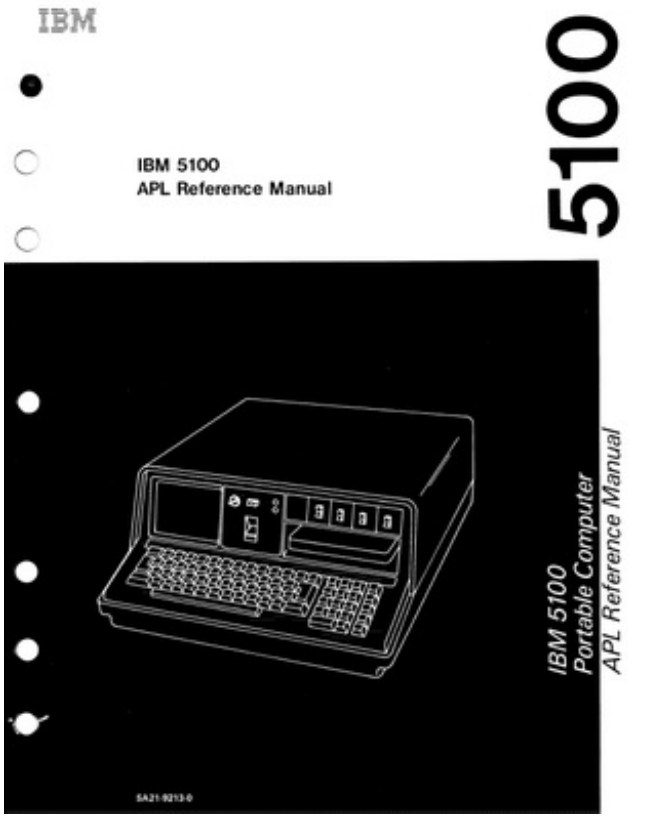
Dynamic Array Dimensions

```
 INTEGER  dataset[ALLOCATABLE](:,:),
+        results[ALLOCATABLE, HUGE](:,:,:)
 INTEGER  reactor, level, calcs, error
 DATA  reactor, level, calcs  / 10, 50, 100 /

 ALLOCATE (dataset(reactor,level),
+        results(reactor,level,calcs), STAT = error)

 IF (error .NE. 0)
+   STOP 'Not enough storage for data; aborting...'
```

Remember the "HUGE" to avoid meltdown

APL

- Array Processing Language
- Dense, terse pile of mathematical operators
- Special characters & keyboard, over-struck



APL Colon

Is ⍠ over-struck : and - and ⍞ ?

No! There's a ÷ key ⍠ is ⍞ and ÷ over-struck

Statement

```
label : expressions A comment
: (colon) Separates a label from the rest of the line.
```

APL Colon 2

Sudoku Solver (K an APL descendent)

```
r:&9#9;c:81#!9;v:(r;c;(_c%3)+3*_r%3);b:+/'a:(-9#0b\:)'!512;f:&:'~a
m:-1_(_.5*)\256;o:a[m]{+/m*x|y}/:\:a;a:(?,/((.=:)'v)@')'+v
o:o,\:512+!9;p:{@[@[x;a y;o z];y;:;512+z]}
g:{$[0>m:|/i:b x;,x;,/g'p[x;i]'f x i?:m]}
G:{(9*!9)_-511+*g p/[81#0;i;-1+x i:&0<x:,/x]}
+/(+/100 10 1*3#*G"I"$/:'1_)'(10*!50)_I:0:`sudoku.txt
```

by Arthur Whitney (via VrAbi on projecteuler)

Forth

- Stack based
- Interweaves compilation, evaluation and interaction
- Low-level. low-resources, bootstrap porting
- Syntax "words" user extensible

Forth Colon

: (colon) defines the following word, entering compilation state up until the following ; (semi-colon)

```
: HELLO  ( -- )  CR ." Hello, world!" ; HELLO
Hello, world!
: X DUP 1+ . . ;
10 X
11 10
```

C

- System programming
- Fast, Direct, Dangerous
- Portable Assembler

C Colon

```c
#include <stdio.h>
int main (int argc, char **argv) {
  int a=0,b=0;
  label: argc < 3 ? 1 : 0;
  switch (argc) {
    case 1: a++; printf("%d %s\n", a, argv[0]);
    case 2: b++; break;
    default:  a++;
  }
  goto label;
}
```

C Colon Digraphs

```
%:include <stdio.h>
int main (int argc, char **argv) <%
  int a=0,b=0;
  label: argc < 3 ? 1 : 0;
  switch (argc) <%
    case 1: a++; printf("%d %s\n", a, argv<:0:>);
    case 2: b++; break;
    default:  a++;
  %>
  goto label;
%>
```

C Colon Output

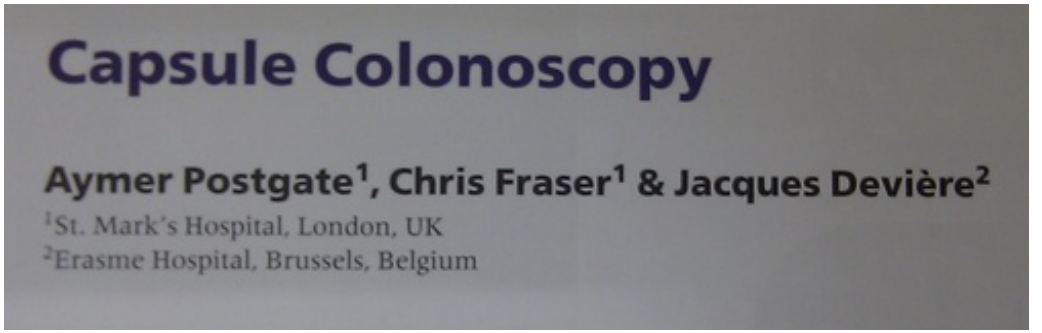
```
$ gcc eg.c -o eg
$ ./eg | head
1 ./eg
2 ./eg
3 ./eg
4 ./eg
5 ./eg
6 ./eg
7 ./eg
8 ./eg
9 ./eg
10 ./eg
```

C Colon Mayhem

```
#include <stdio.h>
#define prime(x) 0 // todo
int main (int x, char **argv) {
  switch (x) {
    default:
    if (prime(x))
      case 4: case 6: case 8: case 9: printf("not ");
    case 2: case 3: case 5: case 7: printf("prime\n");
  }
}
```

Object-Oriented Languages

- Smalltalk
- C++
- Java
  - (I didn’t look at C# but presume it is the same as Java)
- Other OO languages under "Dynamic"
- Go - as a counter-point



Smalltalk

- All values are objects, even classes
- All sending/receiving messages, private state
- Dynamic and reflective

Smalltalk Colon

Assignment

```
vowels := 'aeiou'
```

Chained binary messages ("keyword messages")

```
'hello world' indexOf: $o startingAt: 6
```

Code blocks:

```
[ :params | <message-expressions> ]
[:x | x + 1] value: 3
```

Smalltalk Colon 2

Classes:

```
Object subclass: #MessagePublisher
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Smalltalk Examples'
```

C++

- OO laying siege to C
- Compatible and comparable to C
- Multiple Inheritance
- Operator Overloading
- Generic programming via templates
- Way complicated

C++ Colon

- All of C's uses
- :: is an operator, can't be overloaded (phew)

```
T::X  // Name X defined in class T
N::X  // Name X defined in namespace N
::X   // Global name X
```

C++ Colon 2

Access control, scoping operator

```
class T {     // A new type
  private:    // Accessible only to T's member functions
  protected:  // Also accessable to classes derived from T
  public:     // Accessable to all
  class Z {}; // Nested class T::Z
  T(): x(1) {}// Constructor with initialization list
  T(const T& t): x(t.x) {}  // Copy constructor
}
```

C++ Colon 3

Inheritance access mode

```
class U: public T {};
  // Derived class U inherits all members of base T
class V: private T {};
  // Inherited members of T become private
class W: public T, public U {};
  // Multiple inheritance
class X: public virtual T {};
  // Classes derived from X have base T directly
```

C++ Colon 4

Templates and Namespaces

```
template <class T> X<T>::X(T t) {}
  // Definition of constructor

N::T t;           // Use name T in namespace N
using namespace N;  // Make T visible without N::
```

(Matt Mahoney's C++ Quick Ref)

Java

- C syntax family
- More OO than C++, less than Smalltalk
- Simpler than C++
- JVM, portable and abstracted

Java Colon

- C-like switch
- C-like ? :
- C/Perl-like control labels: for break and continue

Enhanced for loop over collections (> J2SE 5.0)

```
for (int i : intArray) {
  doSomething(i);
}
```

Go

- "Systems programming" modern revision
- Compiled, strongly typed
- Garbage collected and concurrent
- C-ish syntax (control labels, switch)
- Fixes: fall-through, inheritance, pointer math, no ?:
- Interfaces and embedding (vs inheritance)

Go Colon

Short declaration, less type typing

```
t := new(T) // versus var t *T = new(T)
```

Map collection

```
m := map[string]int{"one":1 , "two":2}
```

Go Colon 2

Array slice interface, less pointer arithmetic

```
a[1:3] a[2:] a[:3] a[:]
```

"select" statement for concurrent communication

```
select {
  case i1 = <-c1:
    print("received ", i1, " from c1\n")
}
```

Dynamic Languages

- Perl5
- Python
- Ruby
- JavaScript
- Lua

Perl5

- Pathologically Eclectic mix of C, sh, awk, Unix, …
- C-like block syntax, but compact
- Dynamic runtime, allocation, types, conversions, eval..
- TIMTOWTDI
- Text processing super-powers
- Sigils and punctuation variables
- OO added to Perl 4
- CPAN culture

Perl5 Colon

```
($test) ? $then : $else; # C-like
LABEL: goto LABEL;       # C-like

next LABEL, last LABEL, redo LABEL ; # TIMTOWDI

Package::Separator;      # C++ like

/(?:.*)/               # Regex (?: ) group w/o capture
/[[:punct:]]/            # POSIX char class
use mod :tag;            # Import group convention
```

PDL uses : in ranges and dimensions (Fortran?)

Perl5 Colon 2

perlvar

```
$:
   The current set of characters after which a string
   may be broken to fill continuation fields (starting
   with ^) in a format. Default is " \n-", to break on
   whitespace or hyphens.
   (Mnemonic: a "colon" in poetry is a part of a line.)
```

Wha..?

Python

- Interpreted, Interactive, Object-Oriented
- Clear syntax
- Indentation
- TIOWTDI

Python Colon

```
class A:                        # class
 def blah(self):                  # def
  ages = { 1:"abcd", 5:"xyz" } # dict
  for n in range(10):
    if (ages.has_key(n)):       # conditional
      print ages[n][1:-1]       # range/slice
    else:
      try: raise A()            # try
      except A: "ok"            # except
A().blah()
```

Ruby

- Perl-like without being C-like
- Dynamic and reflective typing
- Thoroughly OO: 1.0.class.class == Class
- Simple yet flexible syntax (blocks)
- Functional (method chaining, blocks,)
- Trendy

Ruby Colon

```ruby
puts :Y if :a_symbol.class == Symbol # "S"

hash = { :water => 'wet', :fire => 'hot' }
puts hash[:fire] # "hot"

class Person
 attr_reader :name, :age
 def initialize(name, age)
   @name, @age = name, age
 end
end
```

Colonoscopy is not very revealing, try endoscopy

JavaScript

- Client-side: dynamic, safe, IO limited
- Prototype OO (Self)
- C syntax family, via Java and Perl
- Badly named

JavaScript Colon

Associative arrays, hence objects, hence JSON

```
{ "k1": "v1", "k2" : 2, "k3" : function () { "v3" } }
```

C/Java/Perl compatible mistakes

```
? :
switch (e) { case v1: x++; break; default: y++ }
```

Lua

- Small language and footprint
- Embeds nicely
- Table is **the** data structure
- Mechanisms, not policy
- OO and other paradigms
- DIY encapsulation

Lua Colon

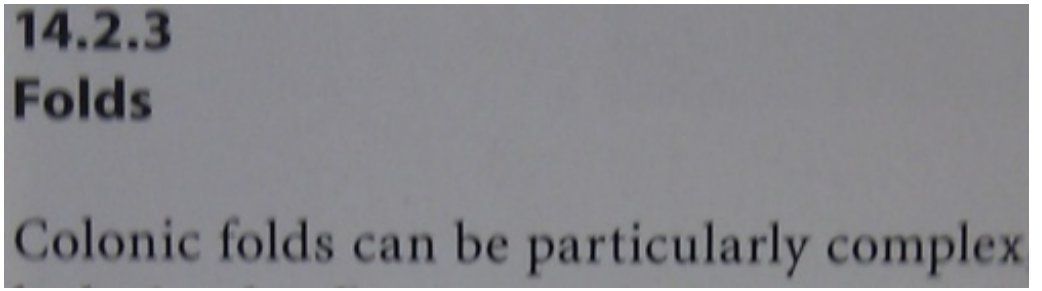
OO syntactic sugar

The colon syntax is used for defining methods, that is, functions that have an implicit extra parameter self.

```
function t.a.b.c:f (params) body end
t.a.b.c.f = function (self, params) body end
t.a.b.c:f(params)  -- call method
```

Functional Languages

- Haskell (ML/OCaml)
- Scala
- Lisp/Scheme/Clojure
- Erlang



Haskell

- Pure Functional Language
- Lazy evaluation
- Strong, inferred typing
- 2WTDI ws/block hs/lhs record/tuple --/{- -}

Haskell Colon

Cons (:) for list construction and matching

```
'a':'b':'c':[] == "abc"
```

Type annotations

```
1             :: (Num t) => t
(1 :: Integer)  :: Integer
(1:)          :: (Num a) => [a] -> [a]
```

Haskell (ML/OCaml) Colon

```
tail :: [a] -> [a]
tail (_:xs) = xs
tail [] =  error "tail"
```

Custom operators from hoogle

```
(:+) (:<) (:>) (:=) (~:)
```

ML/OCaml swap them - : for types and :: for cons

Scala

- OO + Functional hybrid
- JVM and Java integration
- Fancy, inferred static typing
  - Structural typing (duck-ish)
- Mutable/Immutable distinction
  - Helps with concurrency

Scala Colon

Similar to the functional side of the family, adding some typing complications from OO.

```
::      /* List Cons (like ML/OCaml) */
::      /* A class also, apparently */
: Type   /* Type Annotation */
<: >:    /* Covariant and Contravariant types */
```

Scala Colon - Details

```
1 :: 2 :: 3 :: Nil == List(1, 2, 3)
1.::(2.::(3.::(Nil))) // in method calls
```

Operators ending in a colon ':' are right-associative. All other operators are left- associative.

Operators precedence is based on the first character, where *:* gets it's own level

Scala Operators

```
1 +: List(2) :+ 3   // add to collection
List(1) ::: List(2) // concat collections
/: :\ /:\  // colonic folds
:/:       // Document method
=:= <:<    // type constraints
?:        // who knows?
```

See ScalaZ and other libraries for more

Lisp/Scheme/Clojure

- Lisp processing
- Encouraging a functional style
- Anti-syntax s-expressions
- Enabling powerful macros
- Futuristic old school language
- Dynamic
- Garbage collected
- Scheme tidies and tightens
- Clojure - Lisp on JVM with handy data-types
- Adds some of that new-fangled syntax

Lisp/Scheme Colon

"keyword symbols", self-quoting and evaluate to themselves

```
:eof
(defstruct (point (:conc-name nil))
  x y z)
(make-point :x 0 :y 0 :z 200)
;; scheme seems similar
```

Clojure Colon

Maps

```
{:a 1 :b 2}
```

Sets

```
#{:a :b :c}
```

Keywords - symbols starting with : or

```
user> :foo
:foo
user> ::foo
:user/foo
```

Clojure Colon

Three ways to look up a key in a map:

```
(get { :key "value" } :key "default")
(:key { :key "value" })
({ :key "value" } :key)
```

Erlang

- Immutable values, single assignment
- Dynamic typing
- Concurrent and distributed
- Tail call optimized
  - Recursive processes pattern
- Fault-tolerant, hot-swapping
  - Upgrade without downtime

Erlang Colon

Module qualified function names (common)

```
module:function()
```

Hot-swapping

```
-module(a)
loop() ->
  receive
    same_loop ->
      io:format("same~n"),
      loop()
    latest_loop ->
      io:format("latest~n"),
      a:loop()
   end.
```

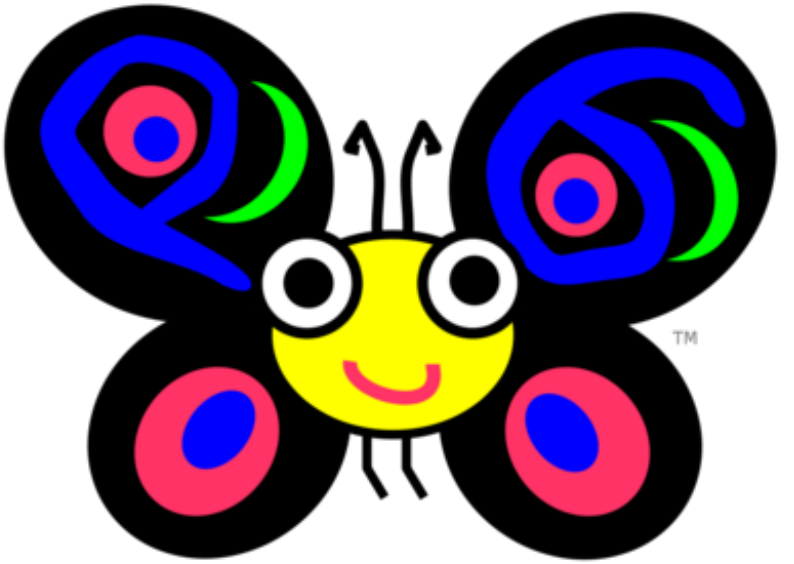
Logic Languages

Prolog Colon

```
head           :- body
sibling(X, Y)  :- parent_child(Z, X), parent_child(Z, Y).
```

Perl 6 - The Second Law Language

- Perl 6 rethinks and reorganizes the Perl 5 patterns
- TIMTWODI - A Maximal Language
- Paradigms: Imperative, OO, Functional, Logical, AOP, DSL
- Hybrid static/dynamic/duck type system
- Lexical, Dynamic and Hypothetical scoping
- OO with roles, reflection and meta-powers
- Next level text processing with Rules and Grammars
- Theoretical answers to questions most people can't ask



Perl 6 - Colon

```
LABEL: next, redo, continue, goto LABEL;

:: # sigil for package/module/class/role/type/grammar/...
$Foo::Bar::baz      # compound identifiers separated by ::
$Foo::($bar)::baz   # ::(...) interpolates symbolic names
$foo $::{'foo'} ::{'$foo'} $::<foo> ::<$foo> # all same
<::($somename)>     # symbolic indirect rule

sub x ($pos, ?$opt, :$named_opt) { say $named_opt }

$:x # twigil, self-declared formal named parameter
{ say "$^a $:b" } # -> $a, :$b { say "$a $b" }
                  # { :::x } a named package param?
```

Perl 6 - Colon Cont’d

```
:adverb, :p, :kv, :a($x), map:{ .say }, q:x 'ls',
%a{b}:exists, s:i:g/this/that/

<foo: 'foo', $bar, 42> # means <foo('foo', $bar, 42)>

:    # Prevents backtracking over previous atom
:?   # Force eager back-tracking on previous atom
:!   # Force greedy back-tracking on previous atom
::   # Fails entire group if previous atom is backtracked
::>  # Discard saved choices in inner alternation "then"
::>: # ::> and : together
:::  # Fails entire rule if previous atom is backtracked
```

Perl 6 - Colon Cont'd

```
# alternate radixes, radii?, radixen?... alternate bases
:10<42> :16<DEAD_BEEF> :60[12,34,56] :2($x)

given $file_handle {
  when :r & :w & :x {...}
  when :!w | :!x    {...}
  when *            {...}
}
```

Perl 6 - Colon Cont'd

```
:(...) # Signature literal, eg. :(Dog $self:)
:=     # Run-time binding $sig := $capture like P5 *A = ..
       # := and .assuming(...) for currying
       # Also used in rules for capturing
::=    # Bind and make read-only, like default sub args
=:=    # Container identity same binding
       # cf. eq == === eqv eqv()
```

```
# Match up named pair values in binding assignment
:(:who($name), :why($reason))
   := (why => $because, who => "me");
my ::MySig ::= :(Int, Num)
   # compile time bind a Signature to a lexical var?
&foo:(Int,Num) # disambiguate which foo of multi
&div:(Int, Int --> Int)
```

Perl 6 - Colon Cont'd

```
sub matchedset (Dog ::T $fido, T $spot) {...} # matching type
sub matchedset (Dog ::T $fido, Dog $spot where T) {...}
```

```
# sub traits
sub x() is ::Foo[...] # definitely a parameterized typename
sub x() is :Foo[...]  # definitely a pair with a list
sub x() is Foo[...]   # depends on whether Foo is a predeclared type
```

Perl 6 - Colon Cont'd

```
infix:<+>    # the official name of the operator in $a + $b
prefix:<+>   # the official name of the operator in +$a
postfix:<--> # the official name of the operator in $a--
circumfix:«<!-- -->»

# Postfix methods .:: .:
$obj.::Class::meth # Class qualified method call
$x.:<++>     # prefix:<++>($x)

# Invocant marker and indirect method calls
:( $self : $x, $y );   # $self is invocant or 1st param
feed $hacker: 'Pizza'; # $hacker.feed('Pizza');
method set_name ($_: $newname) {...}
set_name $obj: "Sam";

[<<R!=:=>>] # reducing-reverse-negated-hyper-equivalence?
```

Adverbial Pair Forms

```
Fat arrow          Adverbial pair  Paren form
=========          ==============  ==========
a => True           :a
a => False          :!a
a => 0             :a(0)
a => $x             :a($x)
a => 'foo'         :a<foo>         :a(<foo>)
a => <foo bar>      :a<foo bar>     :a(<foo bar>)
a => «$foo @bar»    :a«$foo @bar»   :a(«$foo @bar»)
a => {...}         :a{...}         :a({...})
a => [...]         :a[...]         :a([...])
a => $a             :$a
a => @a             :@a
a => %a             :%a
a => &a             :&a
a => @$$a           :@$$a (etc.)
a => %foo<a>        %foo<a>:p
```

Adverbial Pair Gotchas

```
Simple pair         DIFFERS from    which means
===========         ============    ===========
2 => <101010>       :2<101010>      radix literal 0b101010
8 => <123>          :8<123>         radix literal 0o123
16 => <deadbeef>    :16<deadbeef>   radix literal 0xdeadbeef
16 => $somevalue    :16($somevalue) radix conversion function
'' => $x            :($x)           signature literal
'' => ($x,$y)       :($x,$y)        signature literal
'' => <x>           :<x>            name extension
'' => «x»           :«x»            name extension
'' => [$x,$y]       :[$x,$y]        name extension
'' => { .say }      :{ .say }       adverbial block
```

Generalized Quoting - Q:x:qq'$cmd'

```
Short Long       Meaning
===== ====       =======
:x    :exec       Execute as command and return results
:w    :words      Split result on words (no protection) :ww
:q    :single     Interpolate \\, \q and \' (or whatever)
:qq   :double     Interpolate with :s, :a, :h, :f, :c, :b
:s    :scalar     Interpolate $ vars (and :a, :h)
:f    :function   Interpolate & calls
:c    :closure    Interpolate {...} expressions
:b    :backslash  Interpolate \n, \t, etc. (w/ :q at least)
:to   :heredoc    Parse result as heredoc terminator
:regex            Parse as regex
:subst            Parse as substitution
:trans            Parse as transliteration
:code             Quasiquoting
:p    :path       Return a Path object (see S16 for more)
```

Regex and Rule Colon

```
:b :basechar        Match base char ignoring accents, etc
:bytes              Match individual bytes
:c, :continue       Start scanning from string's .pos
:codes              Match individual codepoints
:ex, :exhaustive    Match every possible way (overlapping)
:g, :global         Find all non-overlapping matches
:graphs             Match individual graphemes
:i, :ignorecase     Ignore letter case
:keepall            Recursively force rule to remember all
:chars              Match maximally abstract characters
:nth(N)             Find Nth occurrence. Also 1st, 2nd, 3rd
:once               Only match first time
:p, :pos            Only try to match at string's .pos
:perl5              Use Perl 5 syntax for regex
:ov, :overlap       Match at all possible  positions
:rw                 Claim string for modification
:s, :sigspace       Replaces literal whitespace by \s <?ws>
```

Indirect Object Colon

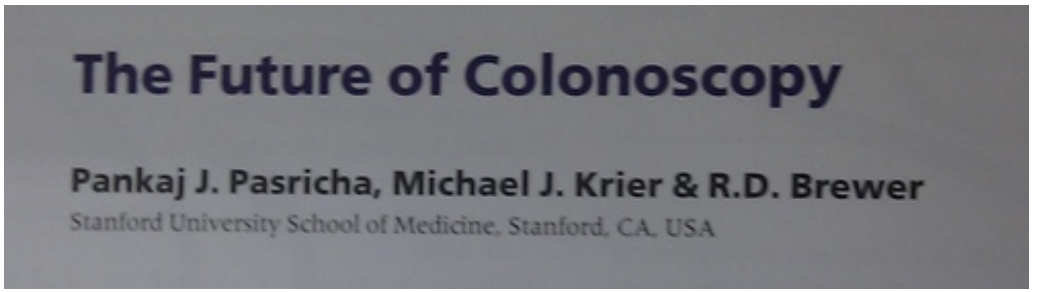
```
foo:               #   label
foo: bar:           #   two labels in a row, okay
.foo: 1            # $_.foo: 1
.$foo: 1           # $_.$foo: 1
foo bar: 1          # bar.foo(1)
foo $bar: 1         # $bar.foo(1)
foo (bar()): 1      # bar().foo(1)
foo .bar:          # foo(.bar:)
foo bar baz: 1      # foo(baz.bar(1))
foo (bar baz): 1    # bar(baz()).foo(1)
```

Indirect/Adverb/Label Colon Parsing Cases

```
foo $obj.bar: 1,2,3     # foo($obj.bar(1,2,3))
foo $obj.bar(): 1,2,3   # foo($obj.bar(1,2,3))
foo $obj.bar(1): 2,3    # foo($obj.bar(1,2,3))
foo $obj.bar(1,2): 3    # foo($obj.bar(1,2,3))
foo($obj.bar): 1,2,3    # foo($obj.bar, 1,2,3)
foo($obj.bar, 1): 2,3   # foo($obj.bar, 1,2,3)
foo($obj.bar, 1,2): 3   # foo($obj.bar, 1,2,3)
foo $obj.bar : 1,2,3    # infix:<:>, $obj.bar.foo(1,2,3)
foo ($obj.bar): 1,2,3   # infix:<:>, $obj.bar.foo(1,2,3)
foo $obj.bar:1,2,3      # syntax error
foo $obj.bar :1,2,3     # syntax error
foo $obj.bar :baz       # adverb, foo($obj.bar(:baz))
foo ($obj.bar) :baz     # adverb, foo($obj.bar, :baz)
foo $obj.bar:baz        # ext. id., foo( $obj.'bar:baz' )
foo $obj.infix:<+>      # ext. id., foo( $obj.'infix:<+>' )
foo: 1,2,3              # label statement start, else infix
```

Colonoscopy Evaluation

- How does the First Law fit various languages?
  - Better than random, but not by much
- Does the use of the colon tell us something about a language?
  - Often, or it can be rationalized
- Can it be used as a quick evaluation of languages?
  - Yeah, sorta, in combination with metrics
  - Gain a little information with very little effort



Colonoscopy Metric

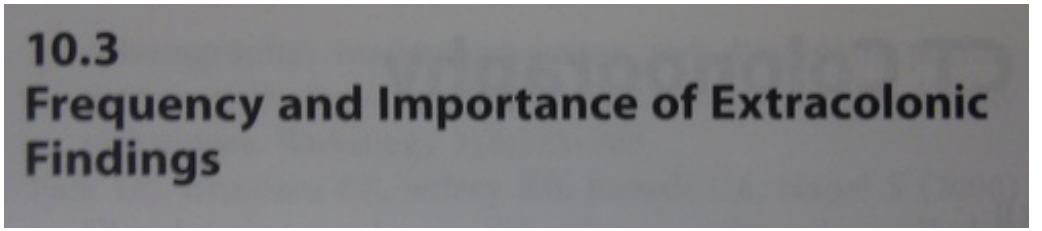
How well does a colonoscopy measure a language?

```
                Units    -10____-5____0____5____10
Colonoscopy                       *
```

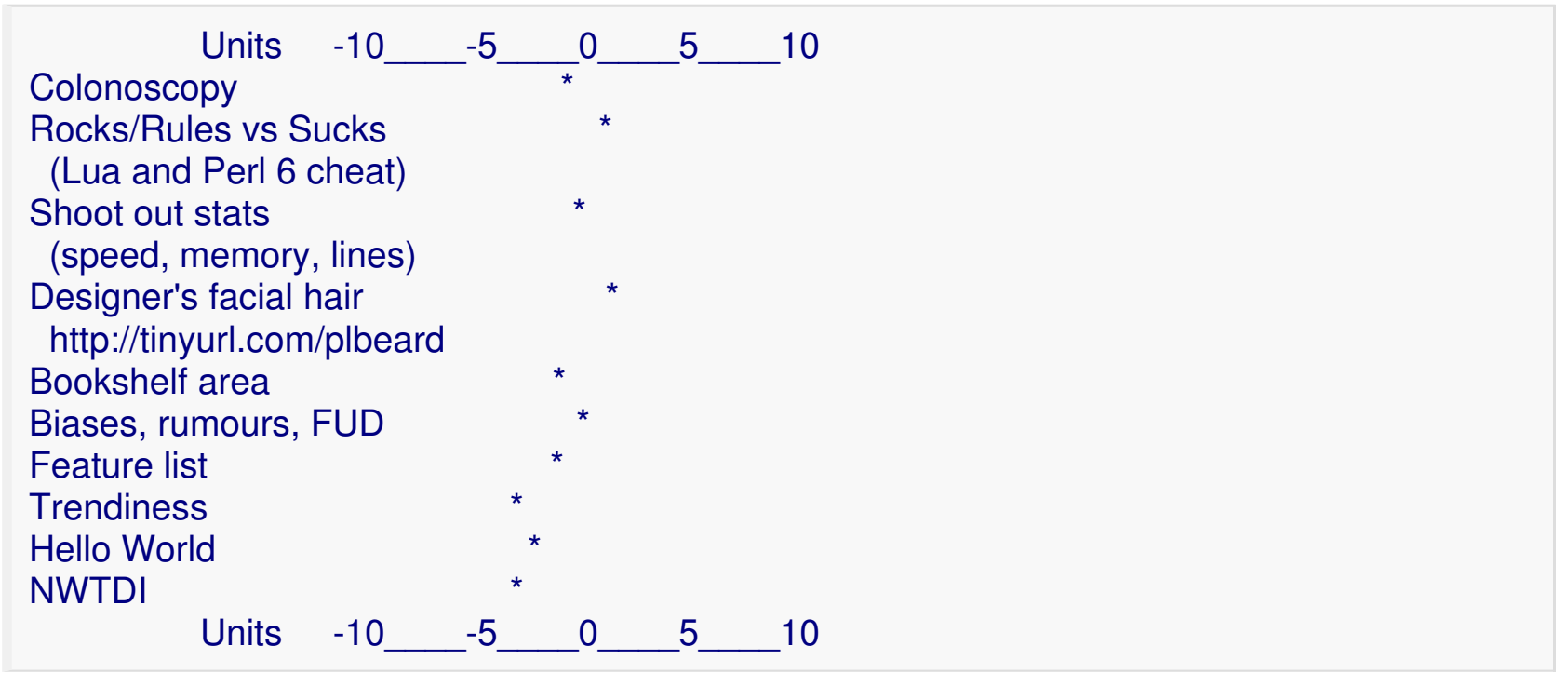
Colonoscopy Metric

How well does a colonoscopy measure a language?

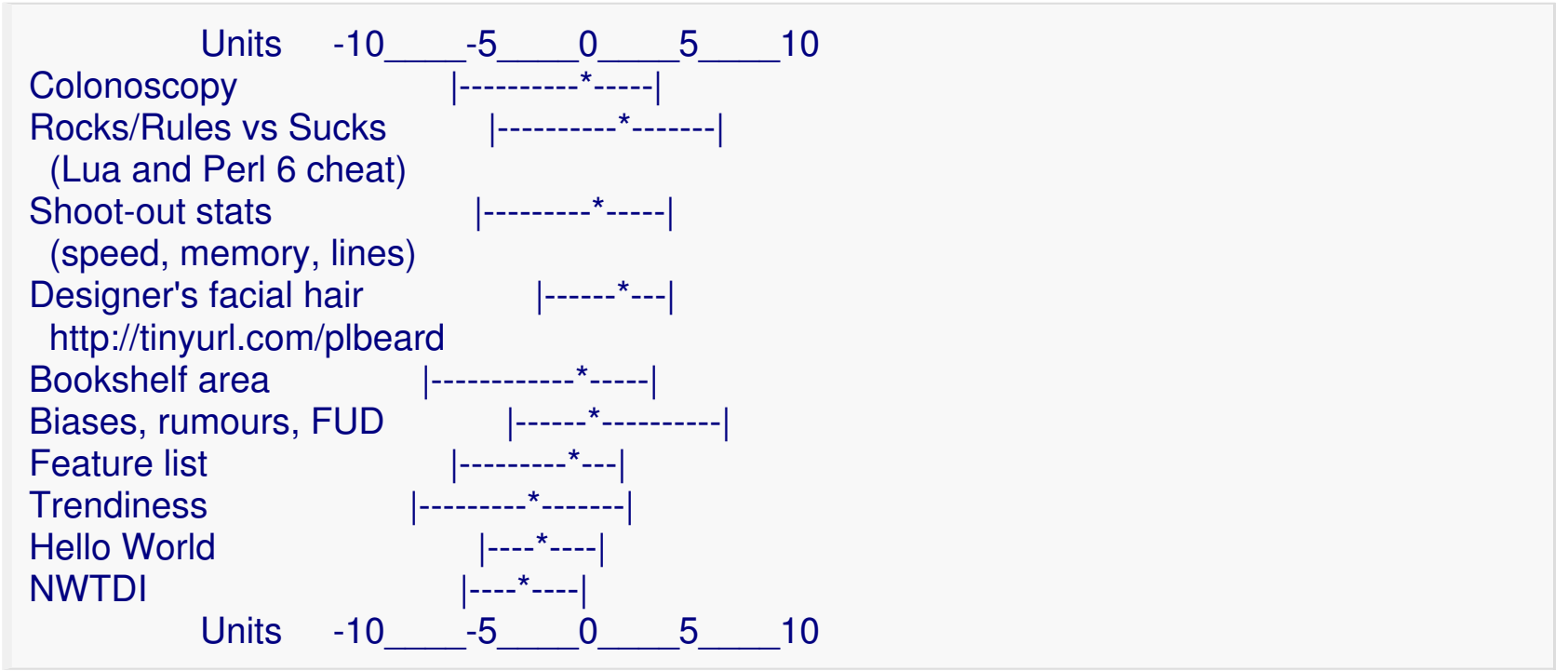
```
            Units    -10____-5____0____5____10
Colonoscopy                      *
                                 ^
                                 |
       Expectations ----------------/
```



Colonoscopy vs Other Metrics



Colonoscopy vs Other Metrics (Error Bars)



End

Brad Bowman

URL: https://github.com/bowman/colonoscopy-talk

Tiny URL: http://tinyurl.com/colonoscopy-talk