Functional Programming You Already Know

@KevlinHenney


SOFTWARE DESIGN PATTERNS
PATTERN-ORIENTED SOFTWARE ARCHITECTURE
A Pattern Language for Distributed Computing
Volume 4
Frank Buschmann
Kevlin Henney
Douglas C. Schmidt
SOFTWARE DESIGN PATTERNS
PATTERN-ORIENTED SOFTWARE ARCHITECTURE
On Patterns and Pattern Languages
Volume 5
Frank Buschmann
Kevlin Henney
Douglas C. Schmidt
Collective Wisdom from the Experts
97 Things Every Programmer Should Know
O'REILLY
Edited by Kevlin Henney

The Paradigms of Programming

Robert W. Floyd
Stanford University

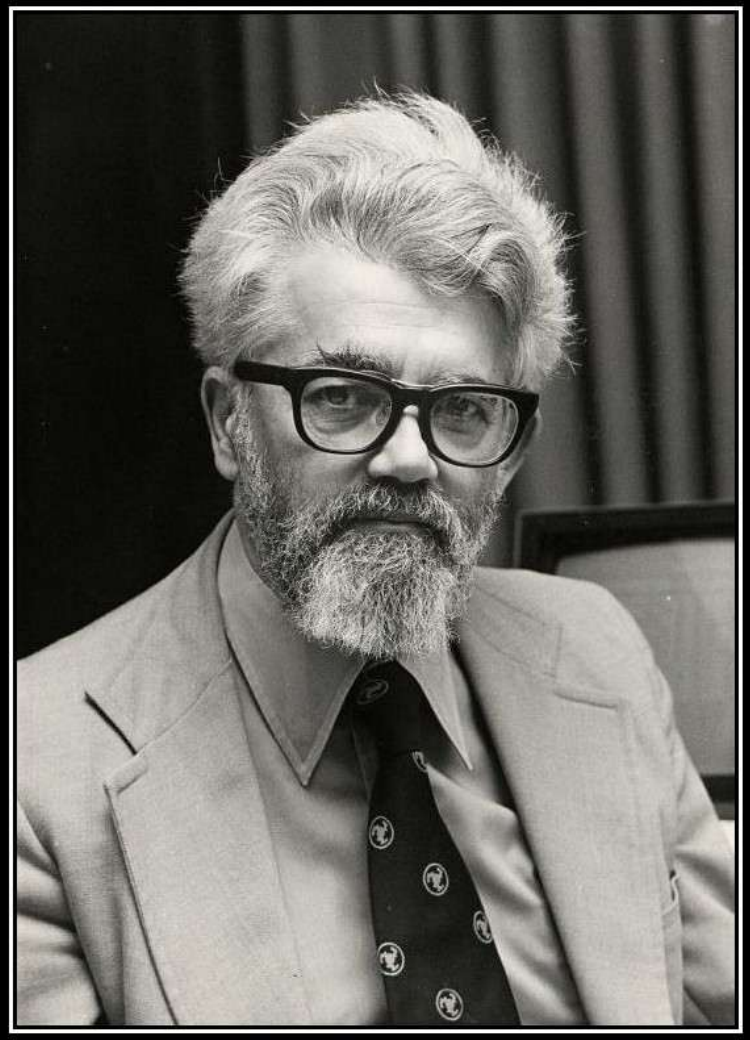
Paradigm(pæ·radim, −dəim) . . . [a. F. *paradigme*, ad. L. *paradigma*, a. Gr. παραδειγμα pattern, example, f. παραδεικνυ·ναι to exhibit beside, show side by side. . .]
1. A pattern, exemplar, example.
1752 J. Gill *Trinity* v. 91
The archetype, paradigm, exemplar, and idea, according to which all things were made.
From the Oxford English Dictionary.

Today I want to talk about the paradigms of programming, how they affect our success as designers of computer programs, how they should be taught, and how they should be embodied in our programming languages.

A familiar example of a paradigm of programming is the technique of *structured programming*, which appears to be the dominant paradigm in most current treatments of programming methodology. Structured programming, as formulated by Dijkstra [6], Wirth [27, 29], and Parnas [21], among others, consists of two phases.

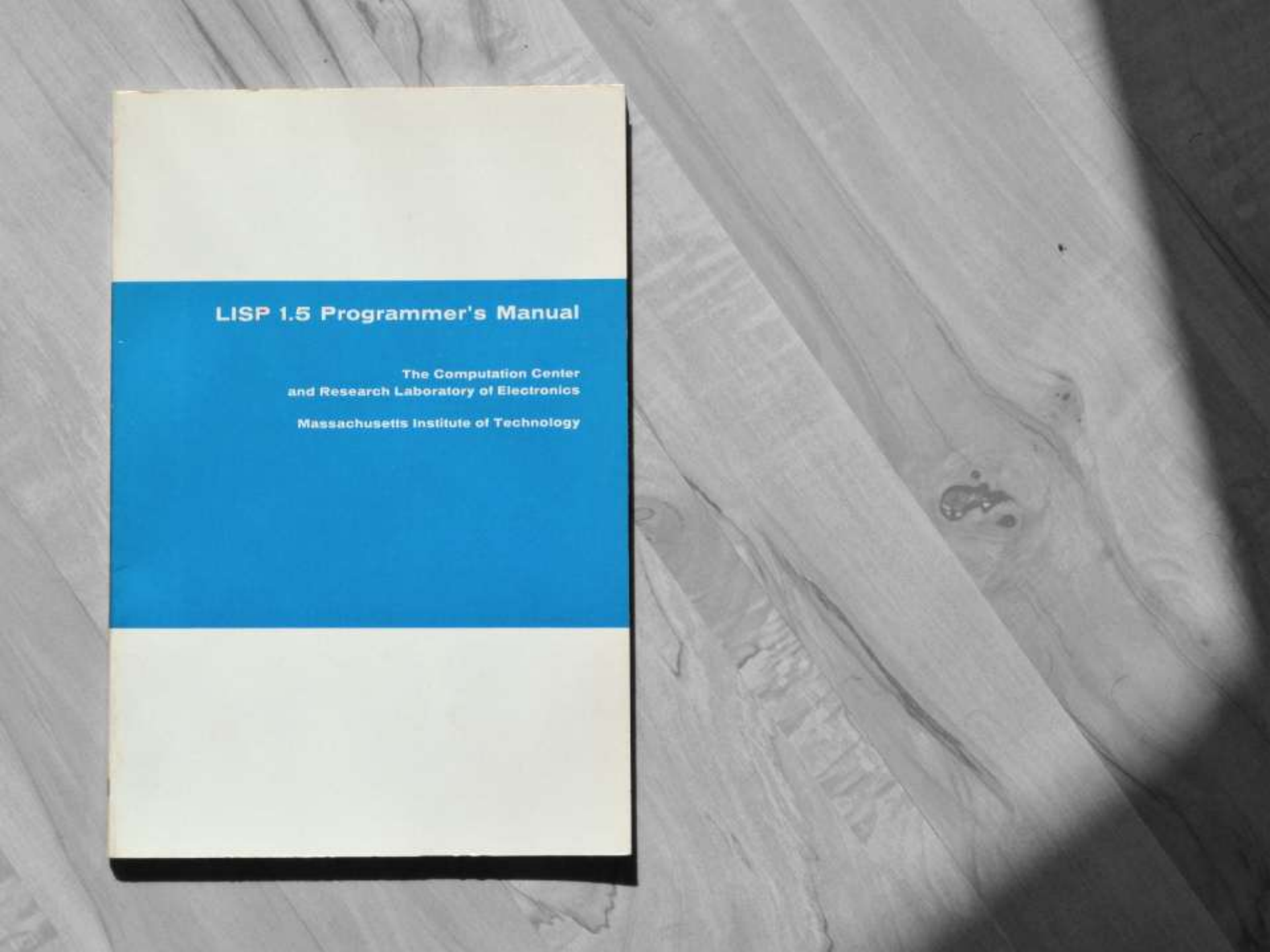
In the first phase, that of top-down design, or stepwise refinement, the problem is decomposed into a very small number of simpler subproblems. In programming the solution of simultaneous linear equations, say, the first level of decomposition would be into a stage of triangularizing the equations and a following stage of back-substitution in the triangularized system. This gradual decomposition is continued until the subproblems that arise are simple enough to cope with directly. In the simultaneous equation example, the back substitution process would be further decomposed as a backwards iteration of a process which finds and stores the value of the *i*th variable from the *i*th equation. Yet further decomposition would yield a fully detailed algorithm.

I believe that the current state of the art of computer programming reflects inadequacies in our stock of paradigms, in our knowledge of existing paradigms, in the way we teach programming paradigms, and in the way our programming languages support, or fail to support, the paradigms of their user communities.



PROGRAMMING

You're Doing It Completely Wrong.

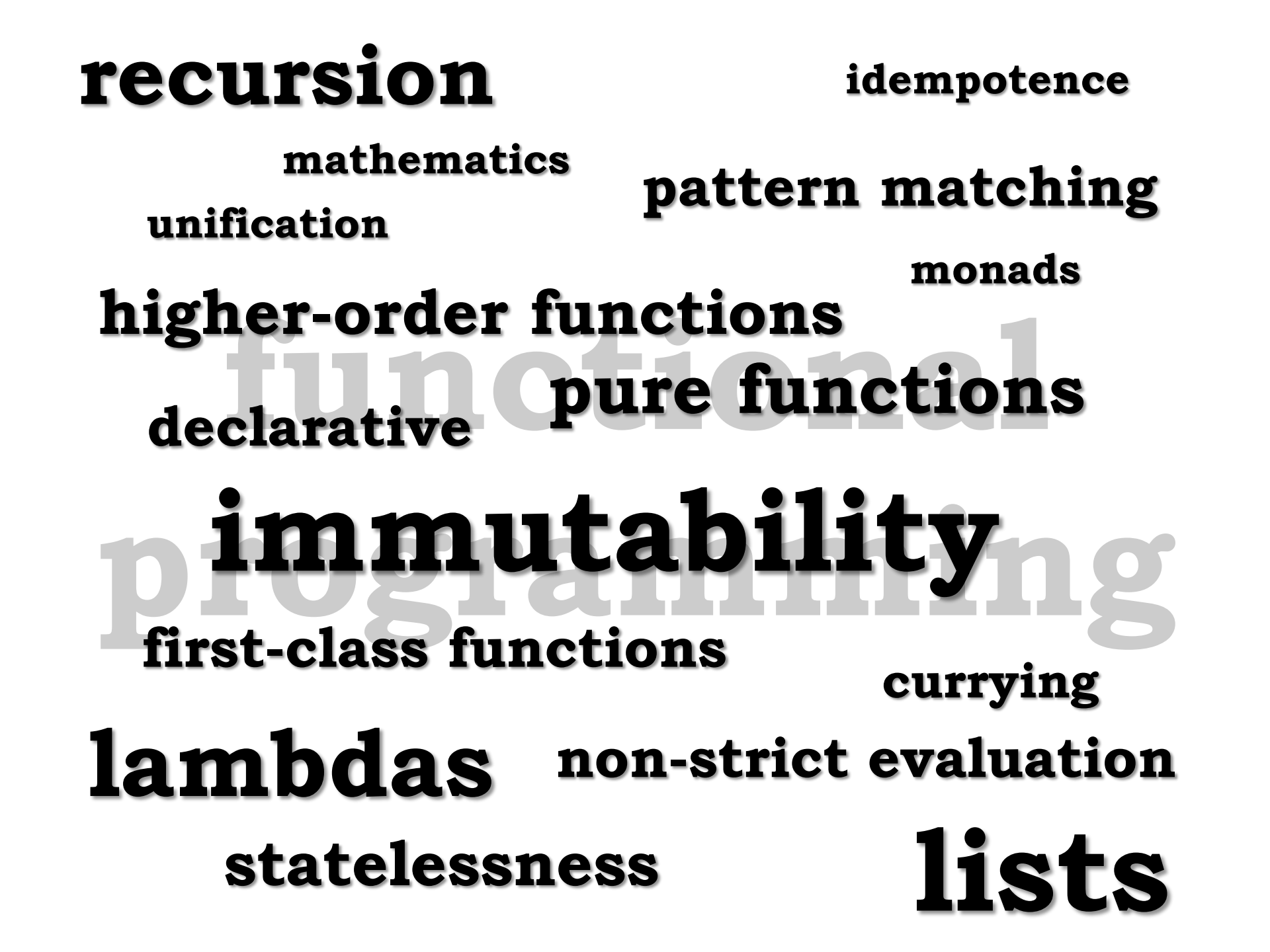

LISP 1.5 Programmer's Manual
The Computation Center
and Research Laboratory of Electronics
Massachusetts Institute of Technology

Unearthing the Excellence in JavaScript

JavaScript: The Good Parts

O'REILLY® | YAHOO! PRESS

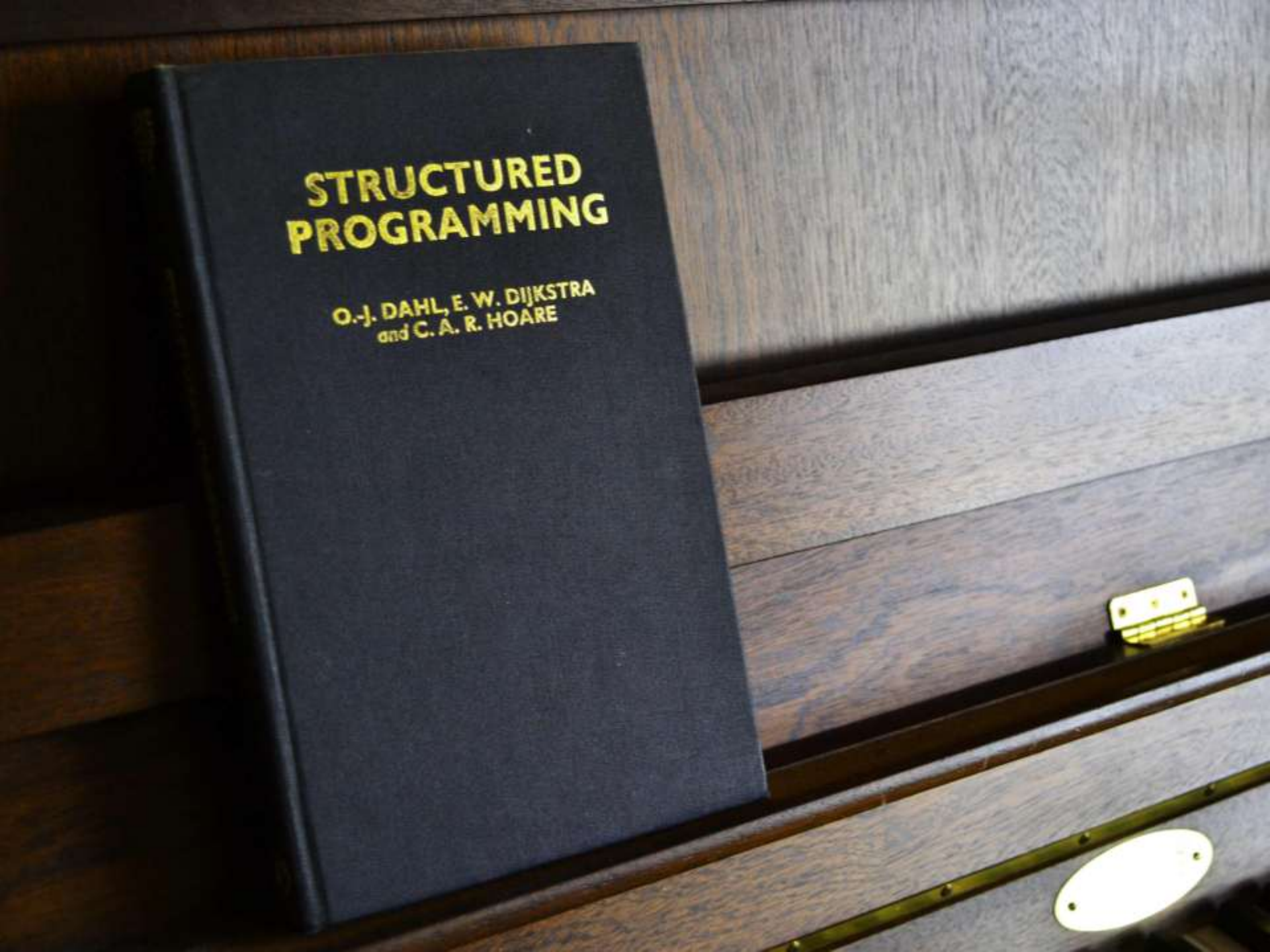
Douglas Crockford


recursion
idempotence
mathematics
pattern matching
unification
monads
higher-order functions
functional
pure functions
declarative
immutability
programming
first-class functions
currying
lambdas
non-strict evaluation
statelessness
lists

```
int atexit(void (*func)(void));
```

```
void qsort(
     void *base,
     size_t nmemb, size_t size,
     int (*compar)(
          const void *, const void *));
```

```
void (*signal(
    int sig, void (*func)(int)))(int);
```


STRUCTURED
PROGRAMMING
O.-J. DAHL, E. W. DIJKSTRA
and C. A. R. HOARE

One of the most powerful mechanisms for program structuring [...] is the block and procedure concept. [...]

A procedure which is capable of giving rise to block instances which survive its call will be known as a class; and the instances will be known as objects of that class. [...]

A call of a class generates a new object of that class.

Ole-Johan Dahl and C A R Hoare
"Hierarchical Program Structures"

```
public class HeatingSystem {
    public void turnOn() …
    public void turnOff() …
    …
}
```

```
public class Timer {
    public Timer(TimeOfDay toExpire, Runnable toDo) …
    public void run() …
    public void cancel() …
    …
}
```

```
public class TurnOn implements Runnable {
    private HeatingSystem toTurnOn;
    public TurnOn(HeatingSystem toRun) {
        toTurnOn = toRun;
    }
    public void run() {
        toTurnOn.turnOn();
    }
}

public class TurnOff implements Runnable {
    private HeatingSystem toTurnOff;
    public TurnOn(HeatingSystem toRun) {
        toTurnOff = toRun;
    }
    public void run() {
        toTurnOff.turnOff();
    }
}
```

```
Timer on =
    new Timer(timeOn, new TurnOn(heatingSystem));
Timer off =
    new Timer(timeOff, new TurnOff(heatingSystem));
```

```
Timer on =
    new Timer(
        timeToTurnOn,
        new Runnable() {
            public void run() {
                heatingSystem.turnOn();
            }
        });
Timer off =
    new Timer(
        timeToTurnOff,
        new Runnable() {
            public void run() {
                heatingSystem.turnOff();
            }
        });
```

```
void turnOn(void * toTurnOn)
{
    static_cast<HeatingSystem *>(toTurnOn)->turnOn();
}

void turnOff(void * toTurnOff)
{
    static_cast<HeatingSystem *>(toTurnOff)->turnOff();
}
```

```
Timer on(timeOn, &heatingSystem, turnOn);
Timer off(timeOff, &heatingSystem, turnOff);
```

```
class Timer
{
public:
    Timer(TimeOfDay toExpire, function<void()> toDo);
    void run();
    void cancel();
    ...
};
```

```
Timer on(
    timeOn,
    bind(&HeatingSystem::turnOn, &heatingSystem));
Timer off(
    timeOff,
    bind(&HeatingSystem::turnOff, &heatingSystem));
```

[ ] ( ) { } ( )

OFFS

```
public class Timer
{
    public Timer(TimeOfDay toExpire, Action toDo) ...
    public void Run() ...
    public void Cancel() ...
    ...
}
```

```
Timer on =
    new Timer(timeOn, () => heatingSystem.TurnOn());
Timer off =
    new Timer(timeOff, () => heatingSystem.TurnOff());
```

```
Timer on = new Timer(timeOn, heatingSystem.TurnOn);
Timer off = new Timer(timeOff, heatingSystem.TurnOff);
```

Lambda-calculus was the first object-oriented language (1941)

William Cook, "On Understanding Data Abstraction, Revisited"

```
newRecentlyUsedList =
  λ • (let items = ref(⟨⟩) •
    {
      isEmpty = λ • #items = 0,
      size = λ • #items,
      add = λ x •
        items := ⟨x⟩^⟨items_y | y ∈ 0...#items ∧ items_y ≠ x⟩,
      get = λ i • items_i
    })
```

```
var newRecentlyUsedList = function() {
    var items = []
    return {
        isEmpty: function() {
            return items.length === 0
        },
        size: function() {
            return items.length
        },
        add: function(newItem) {
            (items = items.filter(function(item) {
                return item !== newItem
            })).unshift(newItem)
        },
        get: function(index) {
            return items[index]
        }
    }
}
```

intension, *n. (Logic)*

- the set of characteristics or properties by which the referent or referents of a given expression is determined; the sense of an expression that determines its reference in every possible world, as opposed to its actual reference. For example, the intension of *prime number* may be *having non-trivial integral factors*, whereas its **extension** would be the set {2, 3, 5, 7, ...}.

E J Borowski and J M Borwein
Dictionary of Mathematics

A list comprehension is a syntactic construct available in some programming languages for creating a list based on existing lists. It follows the form of the mathematical *set-builder notation* (*set comprehension*) as distinct from the use of map and filter functions.

http://en.wikipedia.org/wiki/List_comprehension

$$\{ 2 \cdot x \mid x \in \mathbb{N}, x > 0 \}$$

```
(2 * x for x in count() if x > 0)
```

$$\{ x \mid x \in \mathbb{N}, x > 0 \wedge x \bmod 2 = 0 \}$$

```
(x for x in count() if x > 0 and x % 2 == 0)
```

```
import re, collections

def words(text): return re.findall('[a-z]+', text.lower())

def train(features):
    model = collections.defaultdict(lambda: 1)
    for f in features:
        model[f] += 1
    return model

NWORDS = train(words(file('big.txt').read()))

alphabet = 'abcdefghijklmnopqrstuvwxyz'

def edits1(word):
   splits     = [(word[:i], word[i:]) for i in range(len(word) + 1)]
   deletes    = [a + b[1:] for a, b in splits if b]
   transposes = [a + b[1] + b[0] + b[2:] for a, b in splits if len(b)>1]
   replaces   = [a + c + b[1:] for a, b in splits for c in alphabet if b]
   inserts    = [a + c + b     for a, b in splits for c in alphabet]
   return set(deletes + transposes + replaces + inserts)

def known_edits2(word):
    return set(e2 for e1 in edits1(word) for e2 in edits1(e1) if e2 in NWORDS)

def known(words): return set(w for w in words if w in NWORDS)

def correct(word):
    candidates = known([word]) or known(edits1(word)) or known_edits2(word) or [word]
    return max(candidates, key=NWORDS.get)
```

paraskevidekatriaphobia, *noun*

- The superstitious fear of Friday 13th.
- Contrary to popular myth, this superstition is relatively recent (19th century) and did not originate during or before the medieval times.
- Paraskevidekatriaphobia (or friggatriskaidekaphobia) also reflects a particularly egocentric attributional bias: the universe is prepared to rearrange causality and probability around the believer based on an arbitrary and changeable calendar system, in a way that is sensitive to geography, culture and time zone.


HOLY GOD
WILL BRING
JUDGMENT
DAY ON
MAY 21, 2011
CRY MIGHTILY UNTO
GOD FOR MERCY SEE
PSALMS - 51:
JONAH - 3:

```
struct tm next_friday_13th(const struct tm * after)
{
    struct tm next = *after;
    enum { daily_secs = 24 * 60 * 60 };
    time_t seconds =
        mktime(&next) +
        (next.tm_mday == 13 ? daily_secs : 0);
    do
    {
        seconds += daily_secs;
        next = *localtime(&seconds);
    }
    while(next.tm_mday != 13 || next.tm_wday != 5);
    return next;
}
```

```
std::find_if(
    ++begin, day_iterator(),
    [](const std::tm & day)
    {
        return day.tm_mday == 13 && day.tm_wday == 5;
    });
```

```
class day_iterator : public std::iterator<...>
{
public:
    day_iterator() ...
    explicit day_iterator(const std::tm & start) ...
    const std::tm & operator*() const
    {
        return day;
    }
    const std::tm * operator->() const
    {
        return &day;
    }
    day_iterator & operator++()
    {
        std::time_t seconds = std::mktime(&day) + 24 * 60 * 60;
        day = *std::localtime(&seconds);
        return *this;
    }
    day_iterator operator++(int) ...
    ...
};
```

```
var friday13ths =
    from day in Days.After(start)
    where day.Day == 13
    where day.DayOfWeek == DayOfWeek.Friday
    select day;

foreach(var irrationalBelief in friday13ths)
{
    ...
}
```

```
public class Days : IEnumerable<DateTime>
{
    public static Days After(DateTime startDay)
    {
        return new Days(startDay.AddDays(1));
    }
    public IEnumerator<DateTime> GetEnumerator()
    {
        for(var next = startDay;; next = next.AddDays(1))
            yield return next;
    }
    ...
    private DateTime startDay;
}
```

Concatenative programming is so called because **it uses function *composition* instead of function *application***—a non-concatenative language is thus called *applicative*.

Jon Purdy
http://evincarofautumn.blogspot.in/2012/02/why-concatenative-programming-matters.html

$$f(g(h(x)))$$

$$(f \circ g \circ h)(x)$$

$$x\ h\ g\ f$$

```
function push(top) {
    values[depth++] = top
}
function pop() {
    return values[--depth]
}
function show() {
    for(i = 0; i < depth; ++i)
        printf values[depth - i - 1] " "
    printf "\n"
}

NF > 1          { next }
NF == 0         { show() }
$1 ~ /^[0-9]+$/ { push($1) }
$1 == "+"       { push(pop() + pop()) }
$1 == "*"       { push(pop() * pop()) }
```

$$f \circ g \circ h$$


h | g | f

This is the basic reason Unix pipes are so powerful: they form a rudimentary string-based concatenative programming language.

Jon Purdy
http://evincarofautumn.blogspot.in/2012/02/why-concatenative-programming-matters.html


"After 20 years, this is still the best exposition of the workings of a 'real' operating system."
Ken Thompson
Lions' Commentary on UNIX® 6th Edition
with Source Code
John Lions
Foreword by Dennis Ritchie
"BOOK OF THE YEAR"
Unix Review
1980

Summary--what's most important.

To put my strongest concerns in a nutshell:

1. We should have some ways of coupling programs like garden hose--screw in another segment when it becomes when it becomes necessary to massage data in another way. This is the way of IO also.

2. Our loader should be able to do link-loading and controlled establishment.

3. Our library filing scheme should allow for rather general indexing, responsibility, generations, data path switching.

4. It should be possible to get private system components (all routines are sytem components) for buggering around with.

M. D. McIlroy
Oct. 11, 1964

While Thompson and Ritchie were laying out their file system, McIlroy was "sketching out how to do data processing by connecting together cascades of processes and looking for a kind of prefix-notation language for connecting processes together."

While Thompson and Ritchie were laying out their file system, McIlroy was "sketching out how to do data processing by connecting together cascades of processes and looking for a kind of prefix-notation language for connecting processes together."

Over a period from 1970 to 1972, McIlroy suggested proposal after proposal. He recalls the break-through day: "Then one day, I came up with a syntax for the shell that went along with the piping, and Ken said, I'm gonna do it. He was tired of hearing all this stuff." Thompson didn't do exactly what McIlroy had proposed for the pipe system call, but "invented a slightly better one. That finally got changed once more to what we have today. He put pipes into Unix." Thompson also had to change most of the programs, because up until that time, they couldn't take standard input. There wasn't really a need; they all had file arguments. "GREP had a file argument, CAT had a file argument."

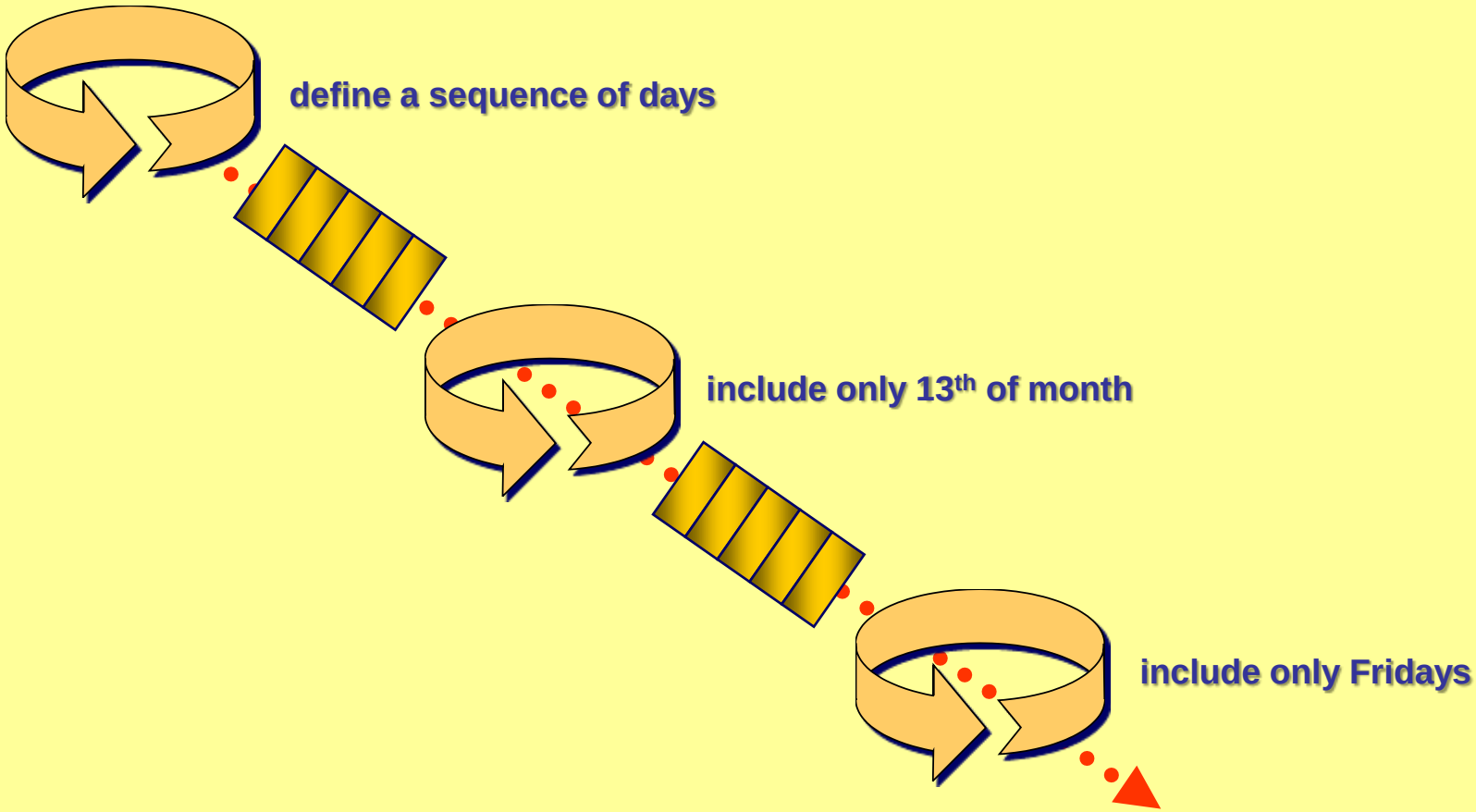
While Thompson and Ritchie were laying out their file system, McIlroy was "sketching out how to do data processing by connecting together cascades of processes and looking for a kind of prefix-notation language for connecting processes together."

Over a period from 1970 to 1972, McIlroy suggested proposal after proposal. He recalls the break-through day: "Then one day, I came up with a syntax for the shell that went along with the piping, and Ken said, I'm gonna do it. He was tired of hearing all this stuff." Thompson didn't do exactly what McIlroy had proposed for the pipe system call, but "invented a slightly better one. That finally got changed once more to what we have today. He put pipes into Unix." Thompson also had to change most of the programs, because up until that time, they couldn't take standard input. There wasn't really a need; they all had file arguments. "GREP had a file argument, CAT had a file argument."

The next morning, "we had this orgy of `one liners.' Everybody had a one liner. Look at this, look at that. ...Everybody started putting forth the UNIX philosophy. Write programs that do one thing and do it well. Write programs to work together. Write programs that handle text streams, because that is a universal interface." Those ideas which add up to the tool approach, were there in some unformed way before pipes, but they really came together afterwards. Pipes became the catalyst for this UNIX philosophy. "The tool thing has turned out to be actually successful. With pipes, many programs could work together, and they could work together at a distance."

In functional programming, a monad is a programming structure that represents computations. Monads are a kind of abstract data type constructor that encapsulate program logic instead of data in the domain model. A defined monad allows the programmer to chain actions together and build different pipelines that process data in various steps, in which each action is decorated with additional processing rules provided by the monad.

http://en.wikipedia.org/wiki/Monad_(functional_programming)


define a sequence of days
include only 13th of month
include only Fridays

```
function GetNextFriday13th($from) {
    [DateTime[]] $friday13ths = &{
        foreach($i in 1..500) {
            $from = $from.AddDays(1)
            $from
        }
    } | ?{
        $_.Day -eq 13
    } | ?{
        $_.DayOfWeek -eq [DayOfWeek]::Friday
    }
    return $friday13ths[0]
}
```

```
[DateTime[][]] $inputsWithExpectations =
    ("2011-01-01", "2011-05-13"),
    ("2011-05-13", "2012-01-13"),
    ("2007-04-01", "2007-04-13"),
    ("2007-04-12", "2007-04-13"),
    ("2007-04-13", "2007-07-13"),
    ("2012-01-01", "2012-01-13"),
    ("2012-01-13", "2012-04-13"),
    ("2012-04-13", "2012-07-13"),
    ("2001-07-13", "2002-09-13")

$inputsWithExpectations | ?{
    [String] $actual = GetNextFriday13th($_[0])
    [String] $expected = $_[1]
    $actual -ne $expected
}
```

Concurrency

Concurrency

Threads

Concurrency

Threads

Locks

Some people, when confronted with a problem, think, "I know, I'll use threads," and then two they hav erpoblesms.

Ned Batchelder

https://twitter.com/#!/nedbat/status/19487382982532710 4

```
public final class Date implements ...
{
    ...
    public int getYear() ...
    public int getMonth() ...
    public int getDayInMonth() ...
    public void setYear(int newYear) ...
    public void setMonth(int newMonth) ...
    public void setDayInMonth(int newDayInMonth) ...
    ...
}
```

```
public final class Date implements ...
{
    ...
    public int getYear() ...
    public int getMonth() ...
    public int getWeekInYear() ...
    public int getDayInYear() ...
    public int getDayInMonth() ...
    public int getDayInWeek() ...
    public void setYear(int newYear) ...
    public void setMonth(int newMonth) ...
    public void setWeekInYear(int newWeek) ...
    public void setDayInYear(int newDayInYear) ...
    public void setDayInMonth(int newDayInMonth) ...
    public void setDayInWeek(int newDayInWeek) ...
    ...
}
```

```
public final class Date implements ...
{
    ...
    public int getYear() ...
    public int getMonth() ...
    public int getWeekInYear() ...
    public int getDayInYear() ...
    public int getDayInMonth() ...
    public int getDayInWeek() ...
    public void setYear(int newYear) ...
    public void setMonth(int newMonth) ...
    public void setWeekInYear(int newWeek) ...
    public void setDayInYear(int newDayInYear) ...
    public void setDayInMonth(int newDayInMonth) ...
    public void setDayInWeek(int newDayInWeek) ...
    ...
    private int year, month, dayInMonth;
}
```

```
public final class Date implements ...
{
    ...
    public int getYear() ...
    public int getMonth() ...
    public int getWeekInYear() ...
    public int getDayInYear() ...
    public int getDayInMonth() ...
    public int getDayInWeek() ...
    public void setYear(int newYear) ...
    public void setMonth(int newMonth) ...
    public void setWeekInYear(int newWeek) ...
    public void setDayInYear(int newDayInYear) ...
    public void setDayInMonth(int newDayInMonth) ...
    public void setDayInWeek(int newDayInWeek) ...
    ...
    private int daysSinceEpoch;
}
```

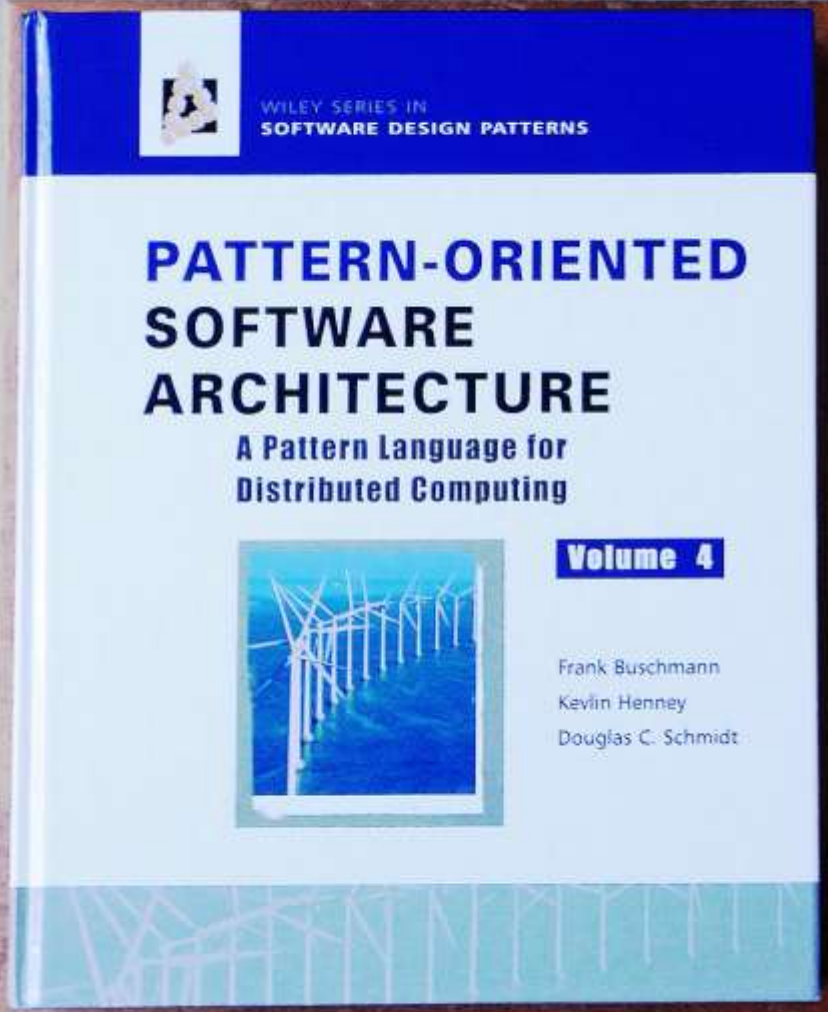
```
public final class Date implements ...
{
    ...
    public int getYear() ...
    public int getMonth() ...
    public int getWeekInYear() ...
    public int getDayInYear() ...
    public int getDayInMonth() ...
    public int getDayInWeek() ...
    ...
}
```

```
public final class Date implements ...
{
    ...
    public int getYear() ...
    public Month getMonth() ...
    public int getWeekInYear() ...
    public int getDayInYear() ...
    public int getDayInMonth() ...
    public DayInWeek getDayInWeek() ...
    ...
}
```

```
public final class Date implements ...
{
    ...
    public int year() ...
    public Month month() ...
    public int weekInYear() ...
    public int dayInYear() ...
    public int dayInMonth() ...
    public DayInWeek dayInWeek() ...
    ...
}
```

Referential transparency and referential opaqueness are properties of parts of computer programs. An expression is said to be referentially transparent if it can be replaced with its value without changing the program (in other words, yielding a program that has the same effects and output on the same input). The opposite term is referentially opaque.

http://en.wikipedia.org/wiki/Referential_transparency_(computer_science)



Immutable Value

Define a value object type whose instances are immutable.

Copied Value

Define a value object type whose instances are copyable.

Instead of using threads and shared memory as our programming model, we can use processes and message passing. *Process* here just means a protected independent state with executing code, not necessarily an operating system process.

Languages such as Erlang (and occam before it) have shown that processes are a very successful mechanism for programming concurrent and parallel systems. Such systems do not have all the synchronization stresses that shared-memory, multithreaded systems have.

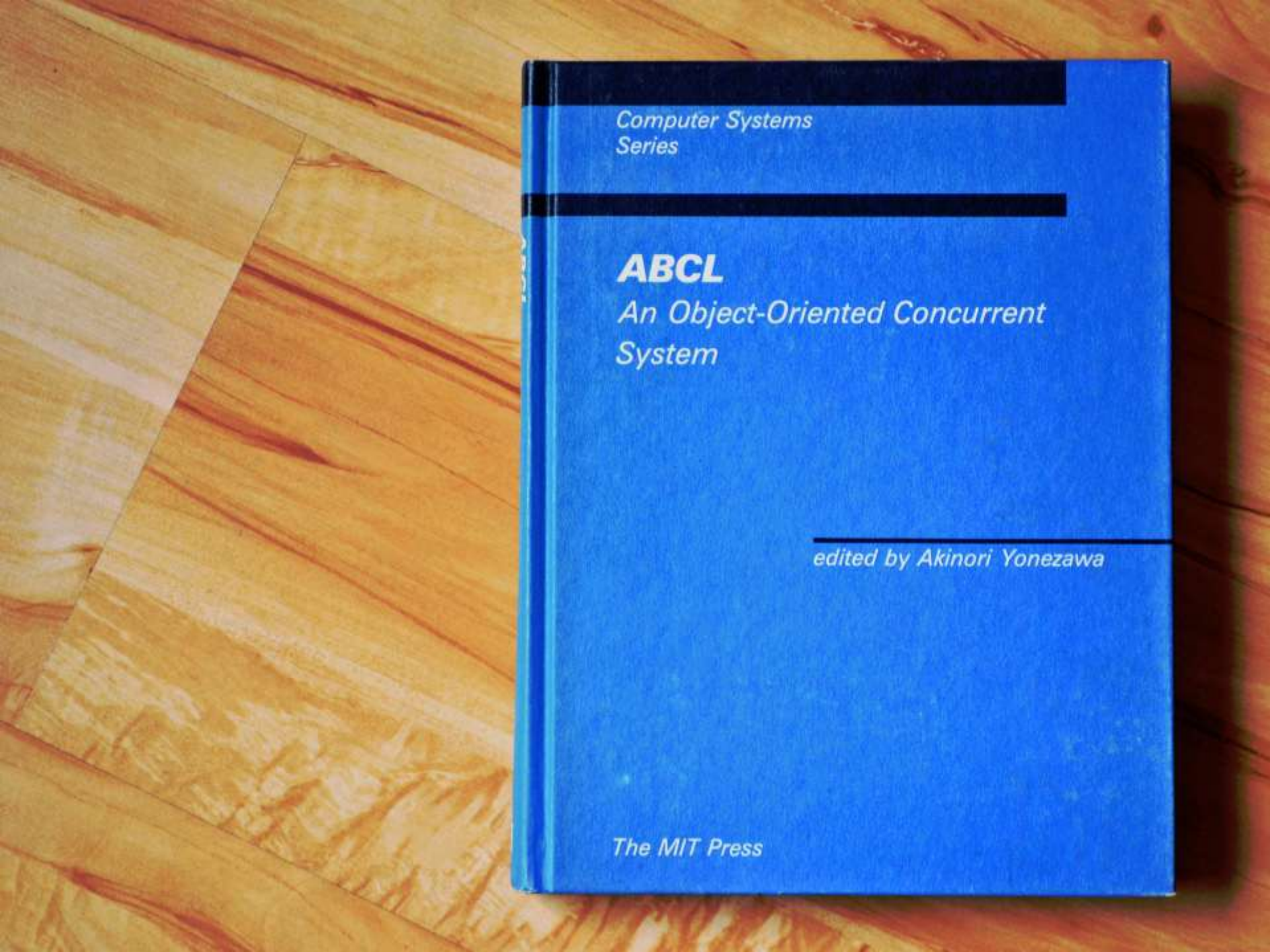
Russel Winder
"Message Passing Leads to Better Scalability in Parallel Systems"
97 Things Every Programmer Should Know

OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things.

Alan Kay

OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things. It can be done in Smalltalk and in LISP. There are possibly other systems in which this is possible, but I'm not aware of them.

Alan Kay


Computer Systems
Series
ABCL
An Object-Oriented Concurrent
System
edited by Akinori Yonezawa
The MIT Press

```
find . -name "*.java" |
sed 's/.*\///' |
sort |
uniq -c |
grep -v "^ *1 " |
sort -r
```

Heinz Kabutz
"Know Your IDE"
97 Things Every Programmer Should Know

```
try {
    Integer.parseInt(time.substring(0, 2));
}
catch (Exception x) {
    return false;
}
if (Integer.parseInt(time.substring(0, 2)) > 12) {
    return false;
}
...
if (!time.substring(9, 11).equals("AM") &
    !time.substring(9, 11).equals("PM")) {
    return false;
}
```

Burk Hufnagel

"Put the Mouse Down and Step Away from the Keyboard"

97 Things Every Programmer Should Know

```
public static boolean validateTime(String time) {
    return time.matches("(0[1-9]|1[0-2]):[0-5][0-9]:[0-5][0-9] ([AP]M)");
}
```

Burk Hufnagel
"Put the Mouse Down and Step Away from the Keyboard"
97 Things Every Programmer Should Know

```
def is_prime(n)
    ("1" * n) !~ /^1?$|^(11+?)\1+$/
end
```

http://www.noulakaz.net/weblog/2007/03/18/a-regular-expression-to-check-for-prime-numbers/

```
$0 % 3 == 0                    { printf "Fizz" }
$0 % 5 == 0                    { printf "Buzz" }
$0 % 3 != 0 && $0 % 5 != 0 { printf $0     }
                               { printf "\n"   }
```

```
echo {1..100} | tr ' ' '\n' | awk '
$0 % 3 == 0                    { printf "Fizz" }
$0 % 5 == 0                    { printf "Buzz" }
$0 % 3 != 0 && $0 % 5 != 0 { printf $0     }
                               { printf "\n"   }
' | diff - expected && echo Pass
```

The makefile language is similar to declarative programming. This class of language, in which necessary end conditions are described but the order in which actions are to be taken is not important, is sometimes confusing to programmers used to imperative programming.

http://en.wikipedia.org/wiki/Make_(software)

```
SELECT time, speaker, title
  FROM Sessions
  WHERE
    date = '2012-04-27' AND
    time >= '11:00:00'
  ORDER BY time
```

```
group ::=
    '(' expression ')'
factor ::=
    integer | group
term ::=
    factor (('*' factor) | ('/' factor))*
expression ::=
    term (('+' term) | ('-' term))*
```

```
group =
    '(' >> expression >> ')';
factor =
    integer | group;
term =
    factor >> *(('*' >> factor) | ('/' >> factor));
expression =
    term >> *(('+' >> term) | ('-' >> term));
```

object
- {}
- { *members* }

members
- *pair*
- *pair* , *members*

pair
- *string* : *value*

array
- []
- [*elements*]

elements
- *value*
- *value* , *elements*

value
- *string*
- *number*
- *object*
- *array*
- **true**
- **false**
- **null**

string
- ""
- " *chars* "

chars
- *char*
- *char chars*

char
- *any-non-control-char*
- \"
- \\
- \/
- \b
- \f
- \n
- \r
- \t
- \u *four-hex-digits*

number
- *integer*
- *integer fraction*
- *integer exponent*
- *integer fraction exponent*

integer
- *digit*
- *non-zero-digit digits*
- - *digit*
- - *non-zero-digit digits*

fraction
- . *digits*

exponent
- *e digits*

e
- e
- e+
- e-
- E
- E+
- E-

To iterate is human,
to recurse divine.

— L Peter Deutsch

```
int factorial(int n)
{
    int result = n;
    while(n-- > 1)
        result *= n;
    return result;
}
```

```
int factorial(int n)
{
    if(n > 1)
        return n * factorial(n - 1);
    else
        return 1;
}
```

```
int factorial(int n, int r)
{
    if(n > 1)
        return factorial(n - 1, n * r);
    else
        return r;
}
```

```
int factorial(int n)
{
    return
        n > 1
        ? n * factorial(n - 1)
        : 1;
}
```

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ (n-1)! \times n & \text{if } n > 0. \end{cases}$$

$$n! = \prod_{k=1}^{n} k$$

seriesProduct(k, k, 1, n)

Excel is the world's most popular functional language.

Simon Peyton Jones

```
#include <stdio.h>

/* printd:  print n in decimal */
void printd(int n)
{
    if (n < 0) {
        putchar('-');
        n = -n;
    }
    if (n / 10)
        printd(n / 10);
    putchar(n % 10 + '0');
}
```

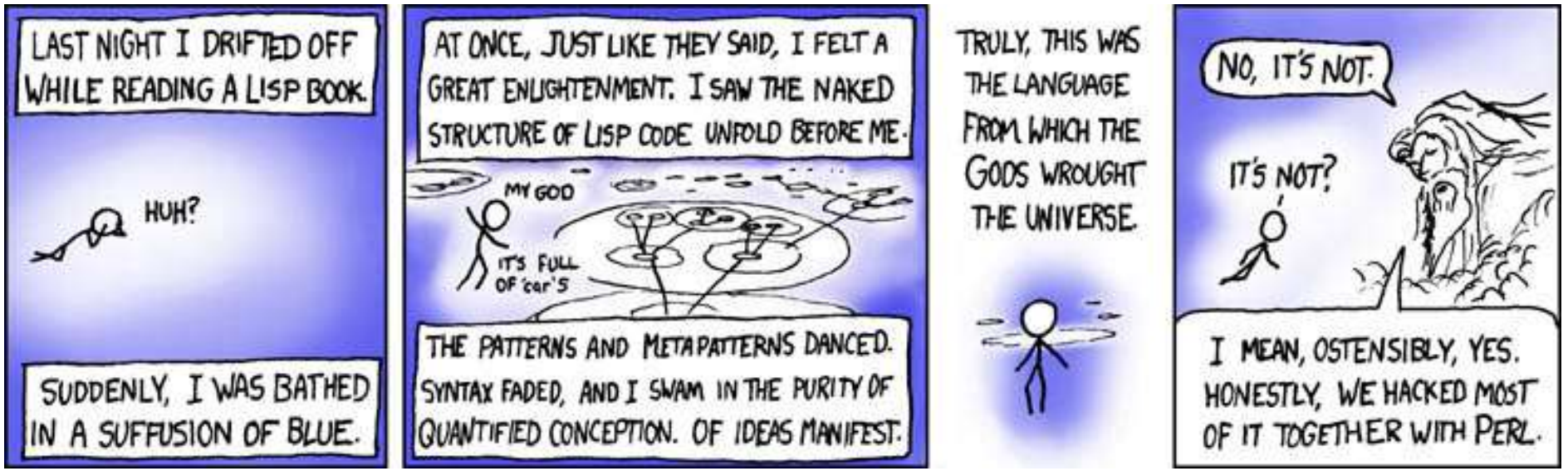
```
/* grep: search for regexp in file */
int grep(char *regexp, FILE *f, char *name)
{
        int n, nmatch;
        char buf[BUFSIZ];

        nmatch = 0;
        while (fgets(buf, sizeof buf, f) != NULL) {
                n = strlen(buf);
                if (n > 0 && buf[n-1] == '\n')
                        buf[n-1] = '\0';
                if (match(regexp, buf)) {
                        nmatch++;
                        if (name != NULL)
                                printf("%s:", name);
                        printf("%s\n", buf);
                }
        }
        return nmatch;
}

/* matchhere: search for regexp at beginning of text */
int matchhere(char *regexp, char *text)
{
        if (regexp[0] == '\0')
                return 1;
        if (regexp[1] == '*')
                return matchstar(regexp[0], regexp+2, text);
        if (regexp[0] == '$' && regexp[1] == '\0')
                return *text == '\0';
        if (*text!='\0' && (regexp[0]=='.' || regexp[0]==*text))
                return matchhere(regexp+1, text+1);
        return 0;
}

/* match: search for regexp anywhere in text */
int match(char *regexp, char *text)
{
        if (regexp[0] == '^')
                return matchhere(regexp+1, text);
        do {    /* must look even if string is empty */
                if (matchhere(regexp, text))
                        return 1;
        } while (*text++ != '\0');
        return 0;
}

/* matchstar: search for c*regexp at beginning of text */
int matchstar(int c, char *regexp, char *text)
{
        do {    /* a * matches zero or more instances */
                if (matchhere(regexp, text))
                        return 1;
        } while (*text != '\0' && (*text++ == c || c == '.'));
        return 0;
}
```

```
// Erwin Unruh, untitled program,
// ANSI X3J16-94-0075/ISO WG21-462, 1994.

template <int i>
struct D
{
    D(void *);
    operator int();
};

template <int p, int i>
struct is_prime
{
    enum { prim = (p%i) && is_prime<(i>2?p:0), i>::prim };
};

template <int i>
struct Prime_print
{
    Prime_print<i-1>    a;
    enum { prim = is_prime<i,i-1>::prim };
    void f() { D<i> d = prim; }
};

struct is_prime<0,0> { enum { prim = 1 }; };
struct is_prime<0,1> { enum { prim = 1 }; };
struct Prime_print<2>
{
    enum { prim = 1 };
    void f() { D<2> d = prim; }
};

void foo()
{
    Prime_print<10> a;
}

// output:
// unruh.cpp 30: conversion from enum to D<2> requested in Prime_print
// unruh.cpp 30: conversion from enum to D<3> requested in Prime_print
// unruh.cpp 30: conversion from enum to D<5> requested in Prime_print
// unruh.cpp 30: conversion from enum to D<7> requested in Prime_print
// unruh.cpp 30: conversion from enum to D<11> requested in Prime_print
// unruh.cpp 30: conversion from enum to D<13> requested in Prime_print
// unruh.cpp 30: conversion from enum to D<17> requested in Prime_print
// unruh.cpp 30: conversion from enum to D<19> requested in Prime_print
```

```
struct Fizz{};
struct Buzz{};
struct FizzBuzz{};

template<int i>
struct RunFizzBuzz
{
   typedef vector<int_<i> > Number;

   typedef typename if_c<(i % 3 == 0) && (i % 5 == 0), FizzBuzz,
                     typename if_c<i % 3 == 0, Fizz,
                              typename if_c<i % 5 == 0, Buzz, Number>::type>::type >::type t1;

   typedef typename push_back<typename RunFizzBuzz<i - 1>::ret, t1>::type ret;
};

template<>
struct RunFizzBuzz<0> // Terminate the recusion.
{
   typedef vector<int_<0> > ret;
};

int main()
{
   typedef RunFizzBuzz<100>::ret::compilation_error_here res;
}
```

```
\Main.cpp(36) : error C2039: 'compilation_error_here' : is not a member of
 'boost::mpl::vector101 <SNIP long argument list>'
    with
    [
          T0=boost::mpl::int_<0>,
          T1=boost::mpl::vector<boost::mpl::int_<1>>,
          T2=boost::mpl::vector<boost::mpl::int_<2>>,
          T3=Fizz,
          T4=boost::mpl::vector<boost::mpl::int_<4>>,
          T5=Buzz,
          T6=Fizz,
          T7=boost::mpl::vector<boost::mpl::int_<7>>,
          T8=boost::mpl::vector<boost::mpl::int_<8>>,
          T9=Fizz,
          T10=Buzz,
          T11=boost::mpl::vector<boost::mpl::int_<11>>,
          T12=Fizz,
          T13=boost::mpl::vector<boost::mpl::int_<13>>,
          T14=boost::mpl::vector<boost::mpl::int_<14>>,
          T15=FizzBuzz,
          <SNIP of elements 16 - 95>
          T96=Fizz,
          T97=boost::mpl::vector<boost::mpl::int_<97>>,
          T98=boost::mpl::vector<boost::mpl::int_<98>>,
          T99=Fizz,
          T100=Buzz
    ]
```

http://www.adampetersen.se/articles/fizzbuzz.htm



We lost the documentation on quantum mechanics. You'll have to decode the regexes yourself.

http://xkcd.com/224/