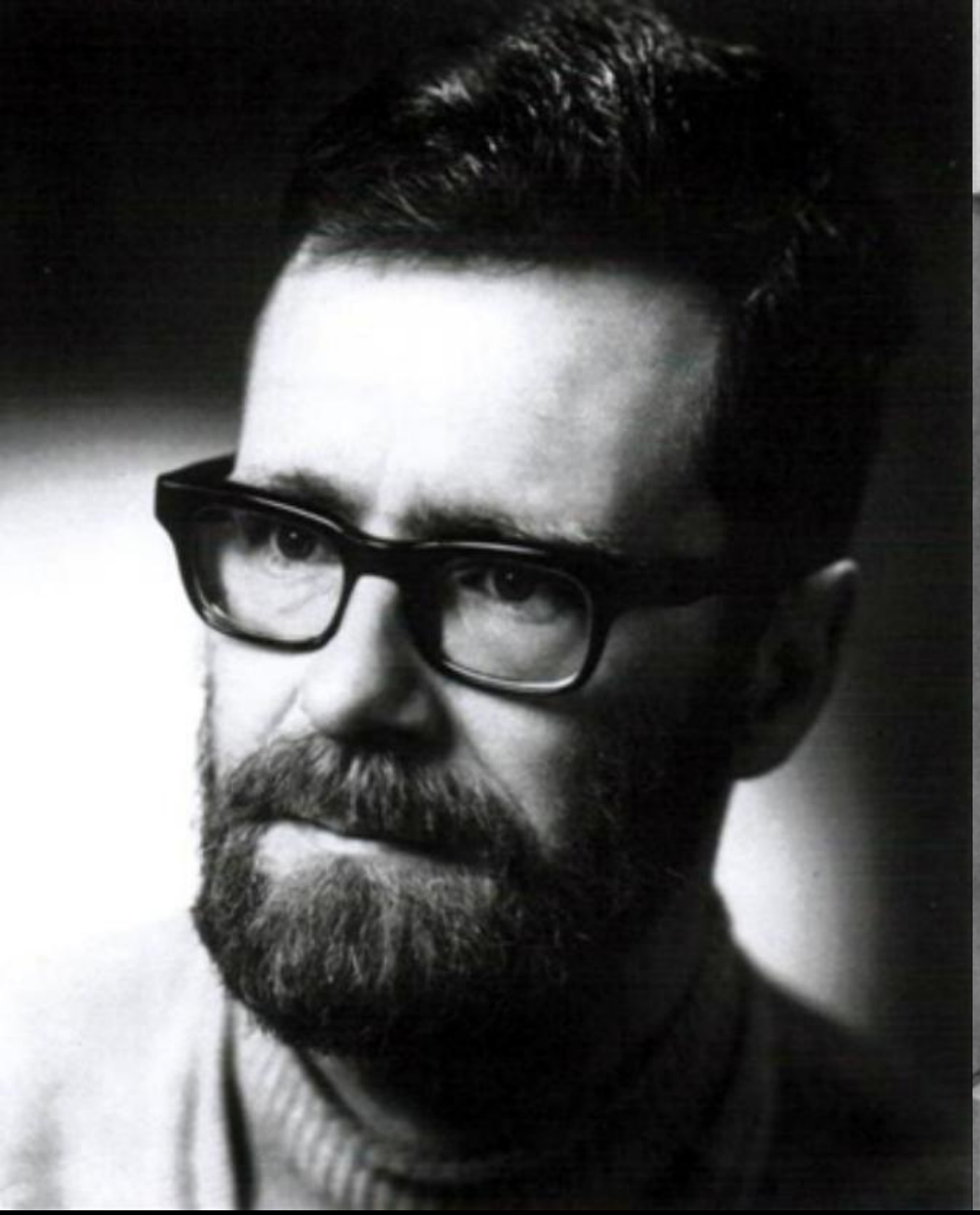

module.exports = class RackApplication
# Create a `RackApplication` for the given configuration and
# root path. The application begins life in the uninitialized
# state.
constructor: (@configuration, @root) ->
@logger = @configuration.getLogger join "apps", basename @root
@readyCallbacks = []
Invoke `callback` if the application's state is ready. Otherwise,
queue the callback to be invoked when the application becomes
ready, then start the initialization process.
ready: (callback) ->
if @state is "ready"
callback()
else
@readyCallbacks.push callback
@initialize()
Stat `tmp/restart.txt` in the application root and invoke the
given callback with a single argument indicating whether or not
the file has been touched since the last call to
`queryRestartFile`.
queryRestartFile: (callback) ->
fs.stat join(@root, "tmp/restart.txt"), (err, stats) =>
if err
@mtime = null
callback false
else
lastMtime = @mtime
@mtime = stats.mtime.getTime()
callback lastMtime isnt @mtime
Collect environment variables from `.powrc` and `.powenv`, in that
order, if present. The idea is that `.powrc` files can be checked
into a source code repository for global configuration, leaving
`.powenv` free for any necessary local overrides.
loadScriptEnvironment: (env, callback) ->
async.reduce [".powrc", ".powenv"], env, (env, filename, callback) =>
exists script = join(@root, filename), (scriptExists) ->
if scriptExists
sourceScriptEnv script, env, callback
else
callback null, env
, callback
If `.rvmrc` and `$HOME/.rvm/scripts/rvm` are present, load rvm,
source `.rvmrc`, and invoke `callback` with the resulting
environment variables. If `.rvmrc` is present but rvm is not

What makes code strange?



minor miracles



Edsger Dijkstra

Donald Knuth

“I do not know of any other technology covering a ratio of 10^10”

– Edsger Dijkstra

Is it bigger than a breadbox?

“By evoking the need for deep conceptual hierarchies, the automatic computer confronts us with a radically new intellectual challenge that has no precedent in our history.”

– Edsger Dijkstra

Code as a Fundamentally New Thing

Code as Logic

```
irb> "code".unpack('b*')

=>

["11000110111101100010011010100110"]
```

Knuth’s Definition:

1. Finiteness
2. Definiteness
3. Input
4. Output
5. Effectiveness

Bridging the Semantic Gap

Code
Abstract Syntax Tree
Bytecode
Machine Instructions
Electricity & Gates

Code as Law

```
ball.x %= screen.width
ball.y %= screen.height
```

In the machine, code is the only possible law.

Code as Art

Knuth: the outer paren.

“Science is knowledge which we understand so well that we can teach it to a computer; and if we don’t understand something, it is an art to deal with it.”

– Donald Knuth

“Even the most perfect reproduction of a work of art is lacking in one element: its presence in time and space, its unique existence at the place where it happens to be.”

– Walter Benjamin

Concrete Arts
vs.
Abstract Arts

Code has a
dual audience

Write for the machine
and for the reader

The machine doesn’t care

```
list.each {|item| item.mark }
```

“The competent programmer is fully aware of the strictly limited size of his own skull; therefore he approaches the programming task in full humility [...]”

– Edsger Dijkstra

“I happen to hold a hard-won minority opinion about code bases. In particular I believe, quite staunchly I might add, that the worst thing that can happen to a code base is size.”

– Steve Yegge

The history of programming languages:

How do we address both audiences?

```
ArrayList list = Lists.newArrayList( ... );
Function upcase = new Function() {
public String apply(String str) {
return str.toUpperCase();
}
};
List upperCaseStrings = Lists.transform(list, upcase);
for (String str : upperCaseStrings) {
System.out.println(str);
}
```

vs.

```
list = [ ... ]
list.map {|str| str.upcase }.each {|str| puts str }
```

Dijkstra argues for Anti-Scripting

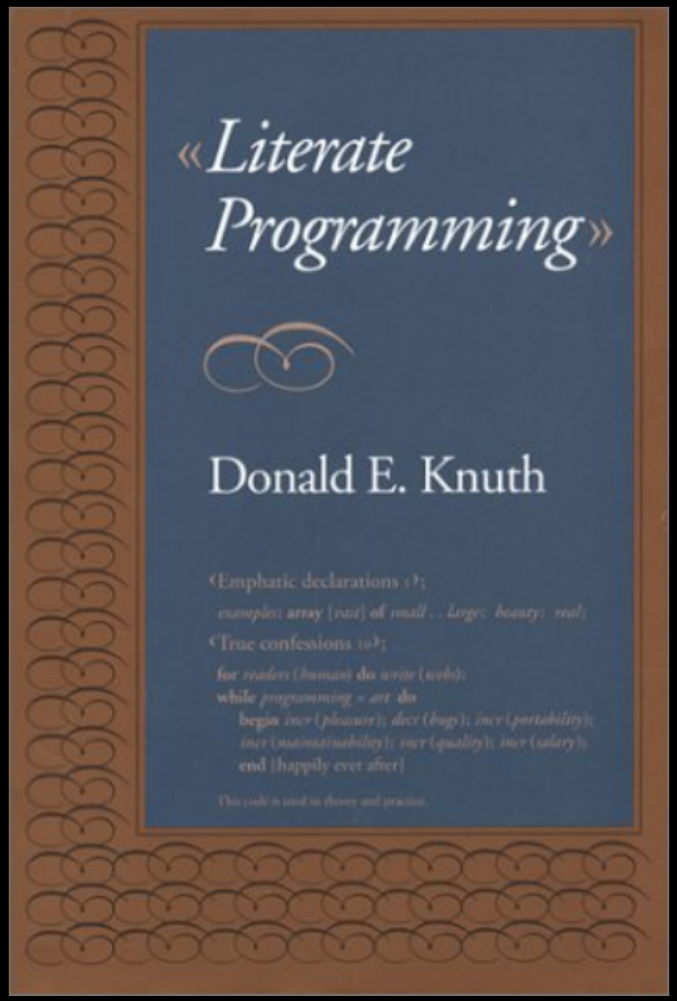
“Sometimes people jot down pseudocode on paper. If that pseudo-code runs directly on their computers, it's best, isn't it? Ruby tries to be like that, like pseudo-code that runs.”

– Yukihiro Matsumoto

Dijkstra's Structured Programming

vs.

Knuth's Literate Programming



Knuth’s Literate Programming

74. The main control loop. Now we've got enough low-level mechanisms in place to start thinking of the program from the top down, and to specify the high-level control.

A global variable *loc* represents where you currently live in the simulated cave. Another variable *newloc* represents where you will go next, unless something like a dwarf blocks you. We also keep track of *oldloc* (the previous value of *loc*) and *oldoldloc* (the previous previous value), for use when you ask to '`go back`'.

```
#define here(t) (toting(t) ∨ place[t] ≡ loc)      /* is object t present? */
#define water_here ((flags[loc] & (liquid + oil)) ≡ liquid)
#define oil_here ((flags[loc] & (liquid + oil)) ≡ liquid + oil)
#define no_liquid_here ((flags[loc] & liquid) ≡ 0)
```

```
⟨ Global variables 7 ⟩ +≡
  location oldoldloc, oldloc, loc, newloc;      /* recent and future locations */
```

75. Here is our overall strategy for administering the game. It is understood that the program might **goto** *quit* from within any of the subsections named here, even though the section names don't mention this explicitly. For example, while checking for interference we might find out that time has run out, or that a dwarf has killed you and no more reincarnations are possible.

The execution consists of two nested loops: There are "minor cycles" inside of "major cycles." Actions define minor cycles in which you stay in the same place and we tell you the result of your action. Motions define major cycles in which you move and we tell you what you can see at the new place.

```
⟨ Simulate an adventure, going to quit when finished 75 ⟩ ≡
  while (1) {
    ⟨ Check for interference with the proposed move to newloc 153 ⟩;
    loc = newloc;      /* hey, we actually moved you */
    ⟨ Possibly move dwarves and the pirate 161 ⟩;
  commence: ⟨ Report the current state 86 ⟩;
    while (1) {
      ⟨ Get user input; goto try_move if motion is requested 76 ⟩;
      ⟨ Perform an action in the current place 79 ⟩;
```

Invoke `callback` if the application's state is ready. Otherwise, queue the callback to be invoked when the application becomes ready, then start the initialization process.

```
ready: (callback) ->
  if @state is "ready"
    callback()
  else
    @readyCallbacks.push callback
    @initialize()
```

Stat `tmp/restart.txt` in the application root and invoke the given callback with a single argument indicating whether or not the file has been touched since the last call to `queryRestartFile`.

```
queryRestartFile: (callback) ->
  fs.stat join(@root, "tmp/restart.txt"), (err, stats) =>
    if err
      @mtime = null
      callback false
    else
      lastMtime = @mtime
      @mtime = stats.mtime.getTime()
      callback lastMtime isnt @mtime
```

Collect environment variables from `.powrc` and `.powenv`, in that order, if present. The idea is that `.powrc` files can be checked into a source code repository for global configuration, leaving `.powenv` free for any necessary local overrides.

```
loadScriptEnvironment: (env, callback) ->
  async.reduce [".powrc", ".powenv"], env, (env, filename, callback) =>
    exists script = join(@root, filename), (scriptExists) ->
      if scriptExists
        sourceScriptEnv script, env, callback
      else
        callback null, env
  , callback
```

If `.rvmrc` and `$HOME/.rvm/scripts/rvm` are present, load rvm, source `.rvmrc`, and invoke `callback` with the resulting environment variables. If `.rvmrc` is present but rvm is not installed, invoke `callback` with an informative error message.

```
loadRvmEnvironment: (env, callback) ->
  exists script = join(@root, ".rvmrc"), (rvmrcExists) =>
    if rvmrcExists
      exists rvm = @configuration.rvmPath, (rvmExists) ->
        if rvmExists
          before = "source '#{rvm}' > /dev/null"
```

Pow

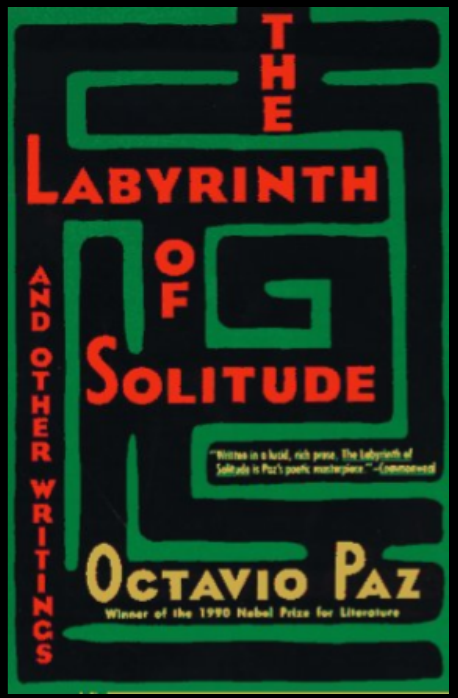
Code as Literature

Meaning belongs to a text or to a program to the extent that it acts upon intelligence in a particular way.

We say that a textual work is “literary” if it contains a high density of meaning.

“In a culture whose already classical dilemma is the hypertrophy of the intellect at the expense of energy and sensual capability, interpretation is the revenge of the intellect upon art.”

– Susan Sontag


THE
LABYRINTH
OF
SOLITUDE
AND OTHER WRITINGS
OCTAVIO PAZ
Winner of the 1990 Nobel Prize for Literature

“While we count the lines of code we use (which are expressed in languages with careful syntax we define), the battles here are fought, won or lost on how much power of meaning lies under the syntactic forms.”

– Alan Kay

Code is (still) Writing

Literary Abstractions in Code

Metaphor

class Circle < Ellipse

Simile

class Rose; include Fragrant

Synecdoche

tree = nodes.root

… and in C

```
int *ptr;
ptr = &array[0];
```

Metonymy

documents.map &:id

Personification

```
# this buffer knows how
# to clear its own cache
```

Object Orientation is a Biological Metaphor

Code as a Commons

“Some programs are elegant, some are exquisite, some are sparkling. My claim is that it is possible to write grand programs, noble programs, truly magnificent ones!”

– Donald Knuth

- "The Work of Art in the Age of Mechanical Reproduction", Walter Benjamin, 1935
- "On the cruelty of really teaching computing science", Edsger Dijkstra
- "The Humble Programmer", Edsger Dijkstra
- "Computer Programming as an Art", Donald Knuth
- "The Philosophy of Ruby", Yukihiro Matsumoto
- "Against Interpretation", Susan Sontag
- "STEPS Toward The Reinvention of Programming", Viewpoints Research Institute
- "Code's Worst Enemy" -- Steve Yegge