“Language Design Patterns”

Tijs van der Storm
storm@cwi.nl / @tvdstorm

Universal Principles of Design

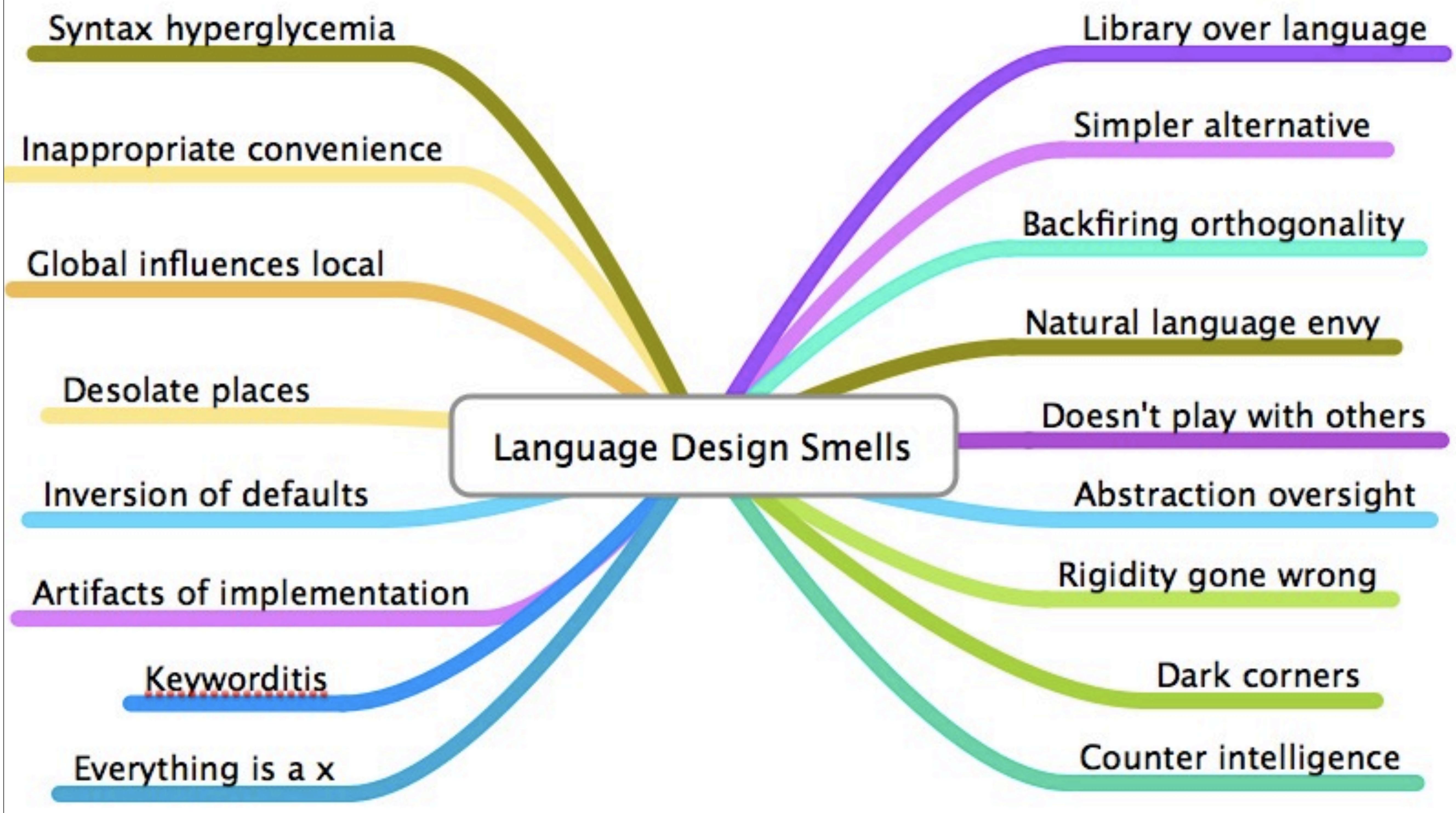
ALIGNMENT
THROPOMORPHIC FORM
ARCHETYPES
AREA ALIGNMENT
ATTRACTIVENESS BIAS
BABY-FACE BIAS
BIOPHILIA EFFECT
CATHEDRAL EFFECT
CHUNKING
CLASSICAL CONDITIONING
CLOSURE
COGNITIVE DISSONANCE
COLOR
COMMON FATE
COMPARISON
CONFIRMATION
CONSISTENCY
CONSTANCY
CONSTRAINT
CONTOUR BIAS
CONTROL
CONVERGENCE
COST-BENEFIT
DEFENSIBLE SPACE
DEPTH OF PROCESSING
DESIGN BY COMMITTEE
DESIRE LINE
DEVELOPMENT CYCLE
ENTRY POINT
ERRORS
EXPECTATION EFFECT
EXPOSURE EFFECT
FACE-ISM RATIO
FACTOR OF SAFETY
FEEDBACK LOOP
FIBONACCI SEQUENCE
FIGURE-GROUND RELATIONSHIP
FITTS' LAW
FIVE HAT RACKS
FLEXIBILITY-USABILITY TRADEOFF
FORGIVENESS
FORM FOLLOWS FUNCTION
FRAMING
FREEZE-FLIGHT-FIGHT-FORFEIT
GARBAGE IN-GARBAGE OUT
GOLDEN RATIO
GOOD CONTINUATION
GUTENBERG DIAGRAM
HICK'S LAW
HIERARCH

REVISED AND UPDATED
25
ADDITIONAL DESIGN PRINCIPLES

William Lidwell
Kritina Holden
Jill Butler

Foreword by
Kimberly Elam

ROCKPORT


Language Design Smells
Syntax hyperglycemia
Inappropriate convenience
Global influences local
Desolate places
Inversion of defaults
Artifacts of implementation
Keyworditis
Everything is a x
Library over language
Simpler alternative
Backfiring orthogonality
Natural language envy
Doesn't play with others
Abstraction oversight
Rigidity gone wrong
Dark corners
Counter intelligence

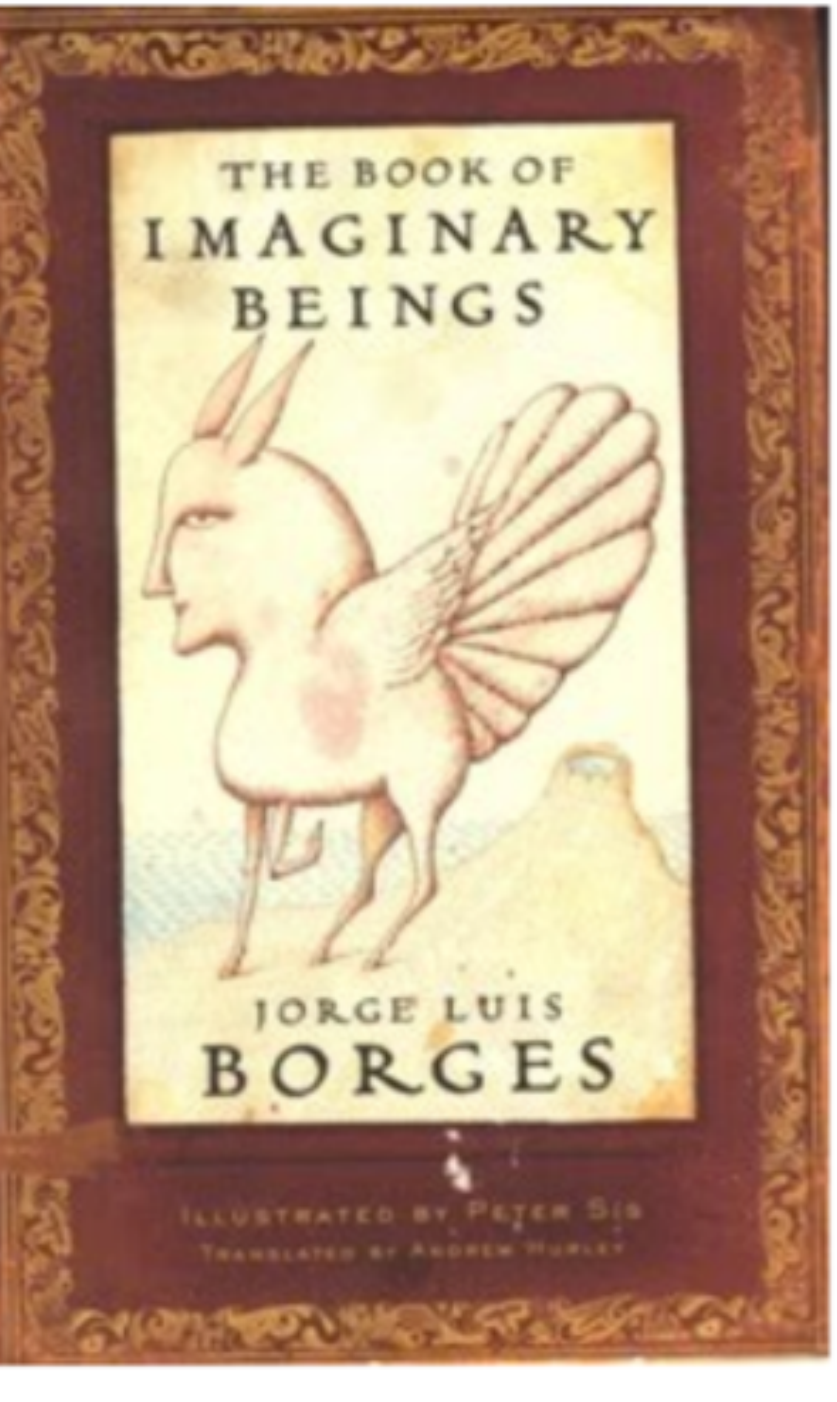
“Language design patterns”

- Patterns?
- Principles?
- Trade-offs?
- Dimensions?
- Tensions?
- Axes?
- Spectra?
- …?

Language design pattern

- A common “reusable” way/guide line to structure/give direction to/highlight aspects of language design
- Making design options explicit
- NOT: language features

Celestial Emporium of Benevolent Knowledge



those that belong to the Emperor,
embalmed ones,
those that are trained,
suckling pigs,
mermaids,
fabulous ones,
stray dogs,
those included in the present classification,
those that tremble as if they were mad,
innumerable ones,
those drawn with a very fine camelhair brush,
others,
those that have just broken a flower vase,
those that from a long way off look like flies.

Flat vs Nesting

- Latex/Html vs WebDSL

```
h1 “Section”
h2 “Subsection”
```

```
section “Section” {
  section “Subsection” {
  }
}
```

- Prolog/Haskell/Rascal/Modula-3/Object Pascal vs. Java etc. -> little redundant but less relative

Run-time does not depend on static (Gilad’s principle)

- Strongtalk
- Counter: Java method overloading, type classes in Haskell

Principle of least surprise (Luke’s remark)

- Take audience into account
- DSLs: notation close to problem domain
- “if it looks like scoping, it should act like scoping”

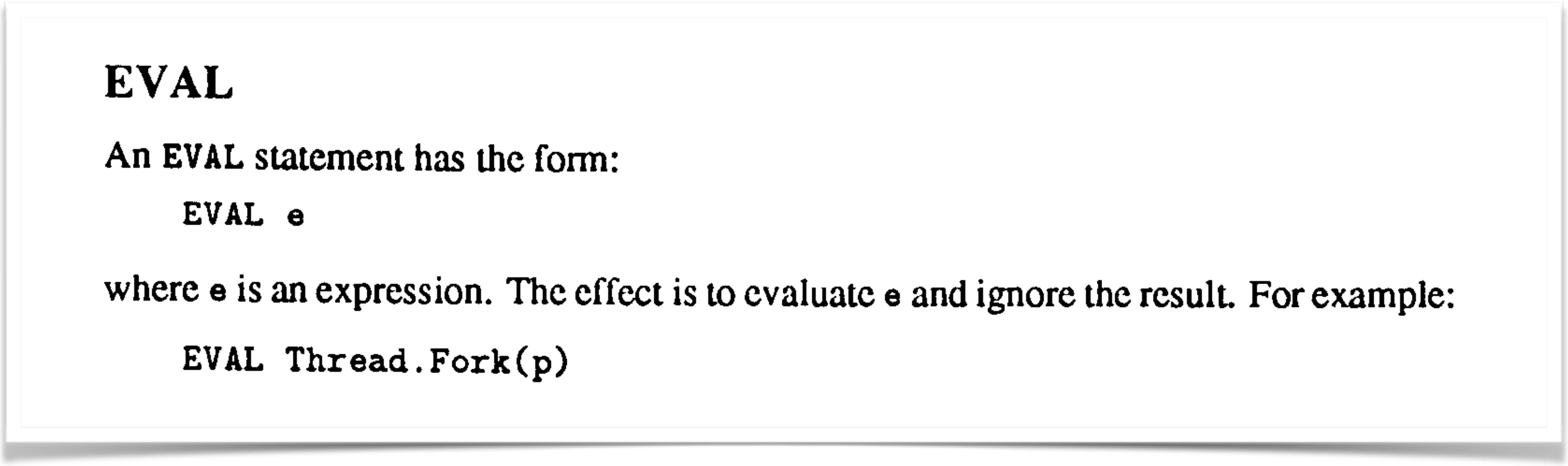
Avoid action at a distance

- Locality, coordinate system between code and execution
- Counter: goto, globals, dynamic scoping
- My favorite counter example: VB6 option base

```java
public class MethodOverloading {

  static class A { }

  static class B extends A { }

  static void foo(A a) {
    System.out.println("A");
  }

  static void foo(B b) {
    System.out.println("B");
  }


  public static void main(String[] args) {
    B b = new B();
    foo(b);
  }

}
```

Explicit vs implicit

- Explicit self, “var”, “EVAL” in Modula-3



EVAL

An `EVAL` statement has the form:

```
    EVAL e
```

where `e` is an expression. The effect is to evaluate `e` and ignore the result. For example:

```
    EVAL Thread.Fork(p)
```

- “put everything in the type system”
- Pony/Self operator (non-)precedence

Sugar and vinegar

- Syntactic sugar: makes for sweeter programming
  - unless x > 1 etc.
- Vinegar: avoid because sour
  - Monads = semantic vinegar

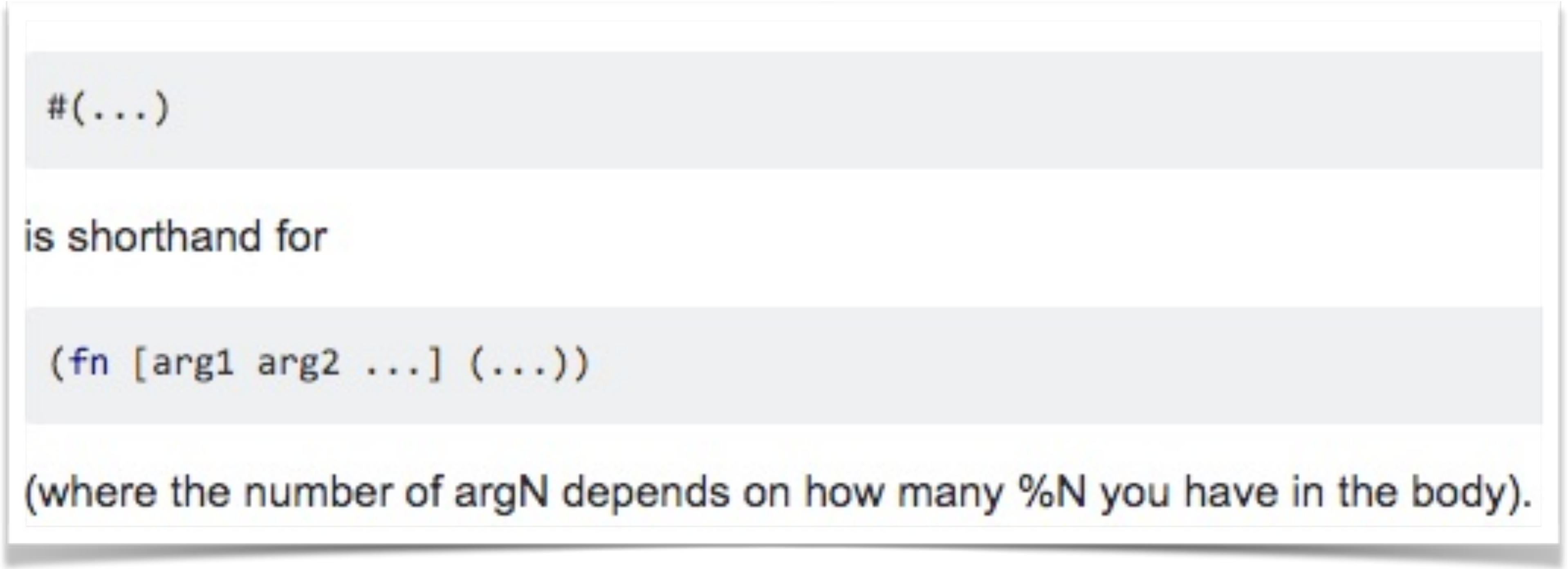
Optimize common path

- “it” in Kotlin/Rascal, Clojure shorthand params

```
( 0 | it + i | i <- [1..100] )
```

- “defn” instead of “define-function” (Arc)

- (Not about efficiency of writing, but “ignorability”)



```
#(...)
```

is shorthand for

```
(fn [arg1 arg2 ...] (...))
```

(where the number of argN depends on how many %N you have in the body).

Manual size as proxy for complexity (@jonathoda)

- “syntax on a business card”
- Eliminate exceptions to the rule
- Small language vs big languages

Discoverability (“dot-driven development”)

- Koka: `x.f(y) = f(x, y)`
- LINQ: `from … select …`

Different things should look different

- Sigils $s, %d (BASIC, Perl, Ruby)
- @field in Ruby
- Counter: writeln and friends in Pascal

Uniformity vs Richness

- Perl 6 vs Self/Smalltalk
- “Everything is a …” thinking
- small language (Scheme) vs big language (CommonLisp)

Orthogonality

- Example: if you have operator overloading, you should support *all* operators

- Algol 68: variables are expressions, and lvalues, so expressions can be lvalues

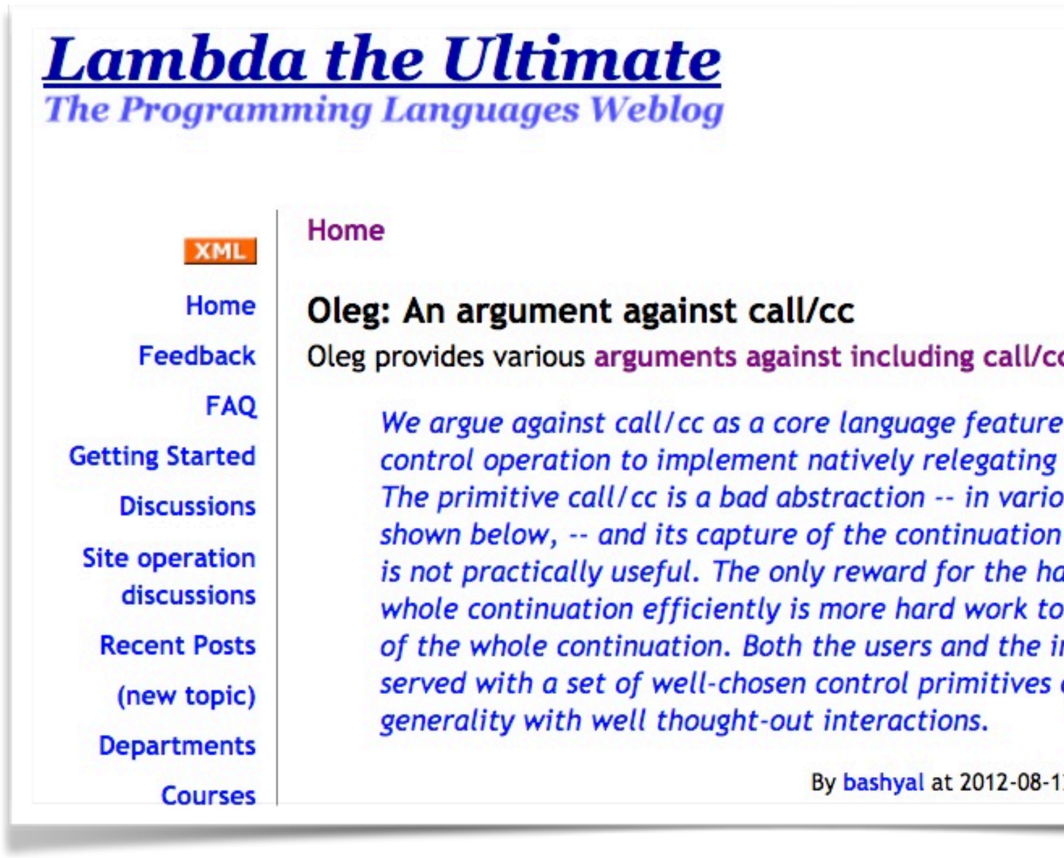
  - if x > 3 then y else z fi := something;

Toolability and implementability

- Trade power for better tooling/faster implementation
- with construct, stack inspection etc.
- co-routines vs call/cc

Fewer degrees of freedom

- Mark Miller "libraries can only make things possible, but never things impossible"
- Static checking, type systems
- DSLs
- fewer degrees of freedom == fewer degrees of freedom to mess up



Oleg: An argument against call/cc

Oleg provides various arguments against including call/cc

We argue against call/cc as a core language feature
control operation to implement natively relegating
The primitive call/cc is a bad abstraction -- in vario
shown below, -- and its capture of the continuation
is not practically useful. The only reward for the ha
whole continuation efficiently is more hard work to
of the whole continuation. Both the users and the i
served with a set of well-chosen control primitives
generality with well thought-out interactions.

By bashyal at 2012-08-1

Semiotics vs Semantics

- EVAL in Modula-3
- ??? in Grace (and Scala?)
- docstrings
- Eiffel contracts
- “JavaDoc”
- @deprecated, override

Minimize line noise/ boilerplate

- `println(“Hello world!”);`
- vs

-
  ```
  public class Main {
    public static void main(…) {
      …
    }
  }
  ```

Language design patterns