Concatenative Clojure

Brandon Bloom

brandon.d.bloom@gmail.com
github.com/brandonbloom
twitter.com/brandonbloom

Concatenative Clojure

Brandon Bloom

brandon.d.bloom@gmail.com
github.com/brandonbloom
twitter.com/brandonbloom

Structure of Talk

- **History Lesson**
  Postfix, stacks, and concatenative
- **Concatenative Clojure**
  Factjor: Concatenative DSL for Clojure
- **Motivating Example**
  DomScript: A stack-based jQuery-like thinggie

Postfix Notation

1950s

Postfix Notation

1950s

- Eliminates grouping via parenthesis

Postfix Notation

1950s

- Eliminates grouping via parenthesis
- Trivial semantics
  - Operands are pushed onto a stack
  - Operators pop operands, push results

Postfix Notation

1950s

- Eliminates grouping via parenthesis
- Trivial semantics
  - Operands are pushed onto a stack
  - Operators pop operands, push results
- Doesn’t necessitate stacks
  - We’ll pretend it does for this talk

Prefix Notation

```
(- 5 2)

(+ (* 10 2) 1)
```

Postfix Notation

```
5 2 -

10 2 * 1 +
```

Postfix Notation

```
5 2 -            =>      3

10 2 * 1 +
```

Postfix Notation

```
5 2 -              =>    3

10 2 * 1 +         =>   21
```

FORTH

1970s

Charles H. Moore

“Lisp is the ultimate high-level language, Forth is the ultimate low-level language”

– Rich Jones

Lisp : Forth ::
Lambdas : Combinators

```
10 5 -                  \ stack: 5
```

```
10 5 -                \ stack: 5
dup *                 \ stack: 25
```

```
10 5 -                 \ stack: 5
dup *                  \ stack: 25
drop                   \ stack empty
```

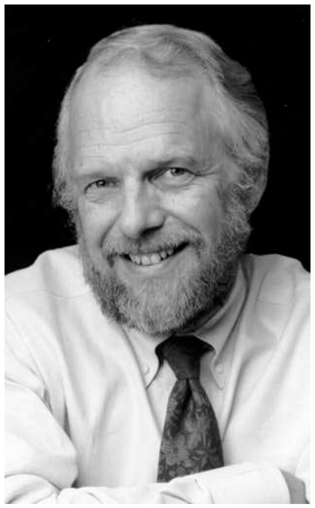
```
10 5 -                 \ stack: 5
dup *                  \ stack: 25
drop                   \ stack empty

: square dup * ;       \ defines square
```

```
10 5 -                  \ stack: 5
dup *                   \ stack: 25
drop                    \ stack empty

: square dup * ;        \ defines square
3 square                \ stack: 9
```

```
10 5 -                  \ stack: 5
dup *                   \ stack: 25
drop                    \ stack empty

: square dup * ;        \ defines square
3 square                \ stack: 9

15 swap -               \ stack: 6
```

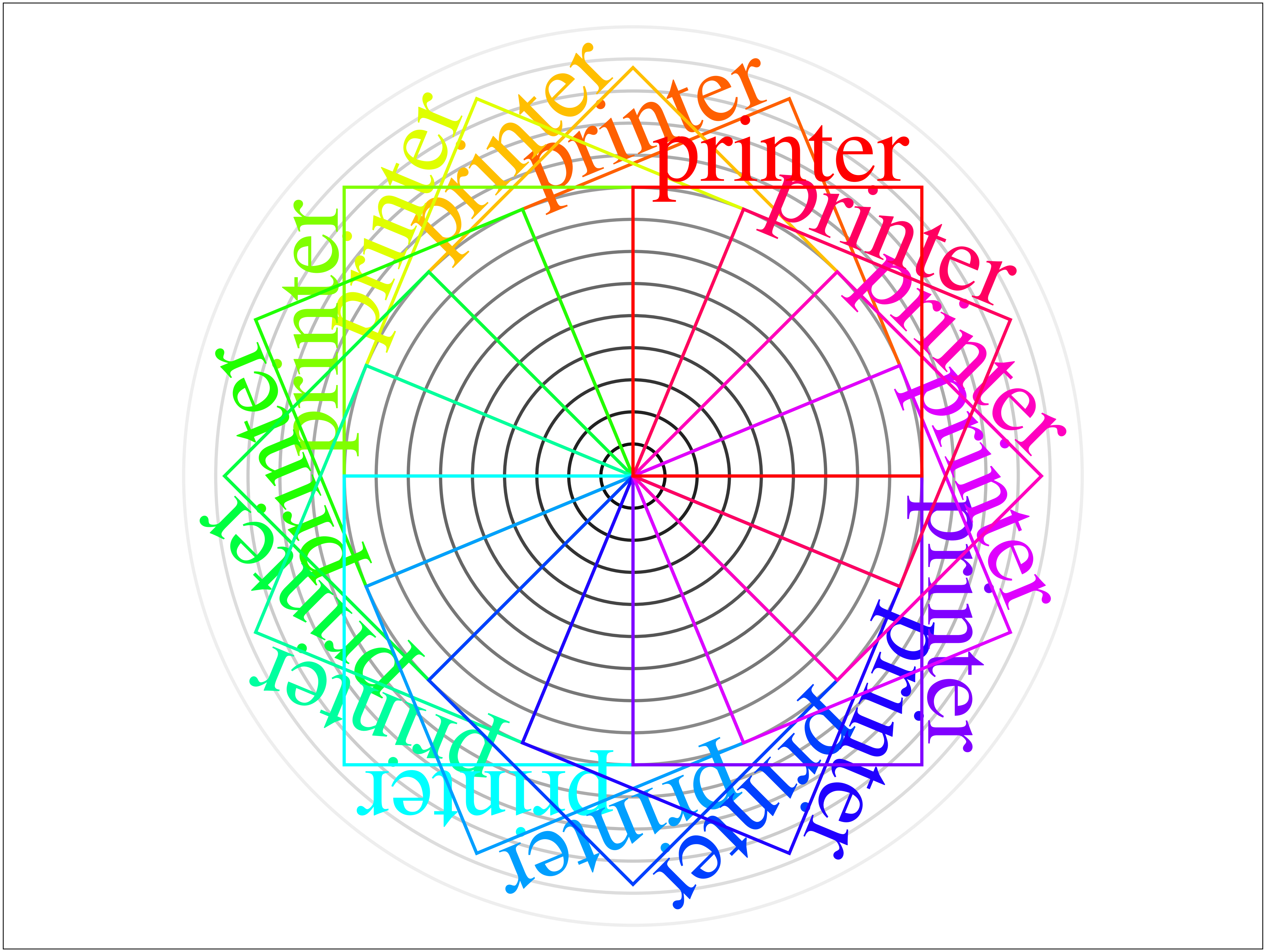
PostScript

(1982)

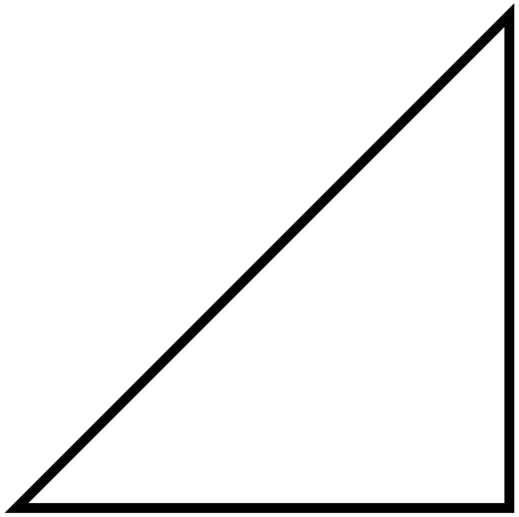


Adobe Co-Founders

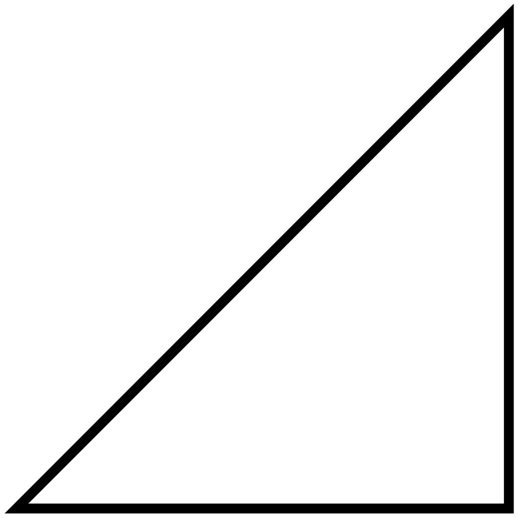
Charles Geschke & John Warnock


printer

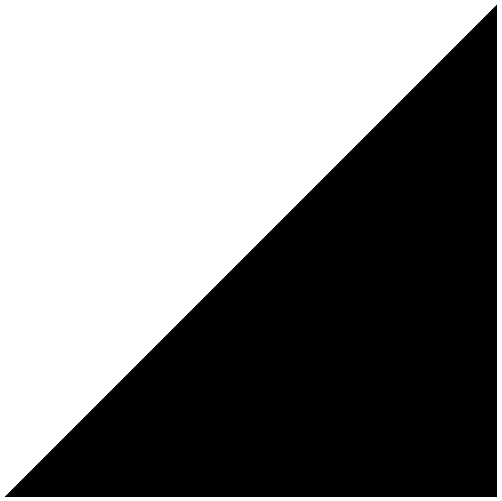
```
% Creates a Triangle path
newpath
 50  50 moveto
300 300 lineto
300  50 lineto
closepath
```



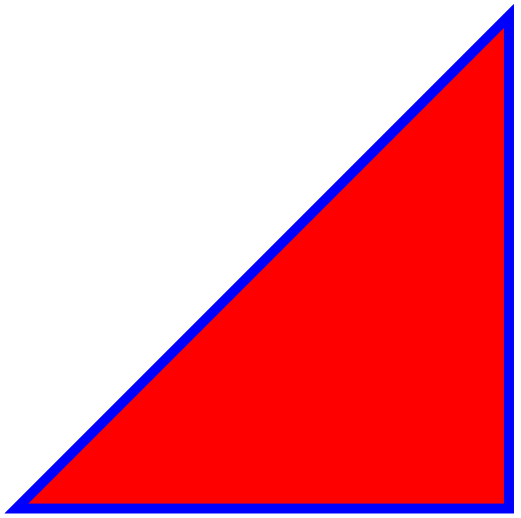
```
% Creates a Triangle path
newpath
 50  50 moveto
300 300 lineto
300  50 lineto
closepath


% Configure pen
5 setlinewidth

% Outline the Triangle
stroke
```



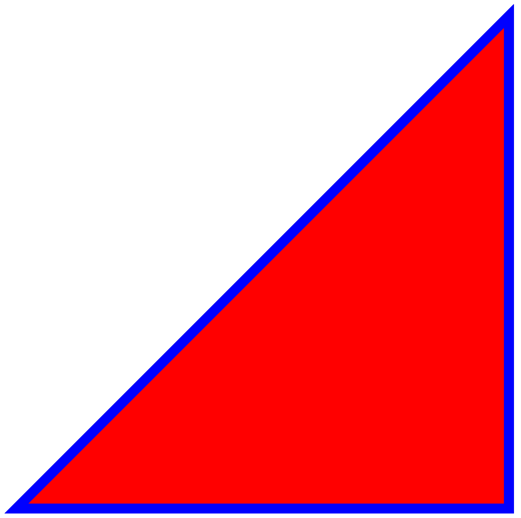
```
% Fill the body of a Triangle
newpath
 50  50 moveto
300 300 lineto
300  50 lineto
closepath
fill
```



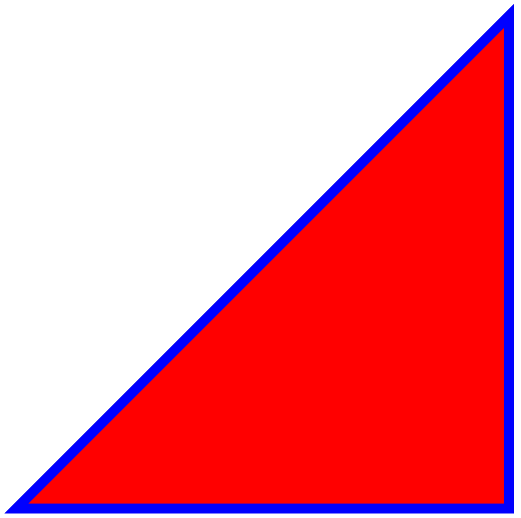
```
% Create a Triangle abstraction
/triangle {
  newpath
   50  50 moveto
  300 300 lineto
  300  50 lineto
  closepath
} def
```



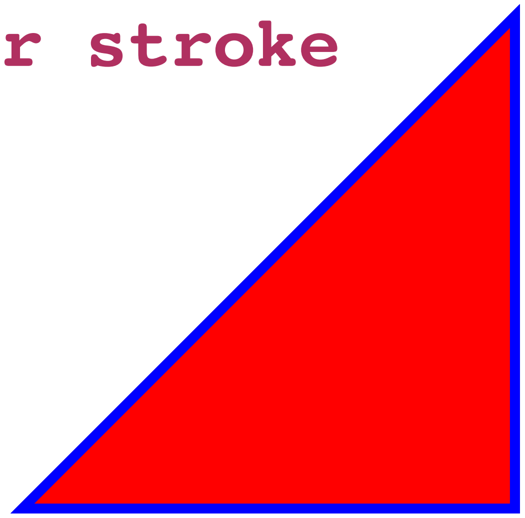
```
% Create a Triangle abstraction
/triangle {
  newpath
   50  50 moveto
  300 300 lineto
  300  50 lineto
  closepath
} def

% Fill and Stroke
1 0 0 setrgbcolor % Red
triangle fill
0 0 1 setrgbcolor % Blue
triangle stroke
```



```
% Don't build path twice
triangle
gsave
1 0 0 setrgbcolor fill
grestore
0 0 1 setrgbcolor stroke
```



```
% Abstract over both stroke and fill
/draw {
  gsave    1 0 0 setrgbcolor fill
  grestore 0 0 1 setrgbcolor stroke
} def

% Draw a full triangle
triangle draw
```



Left-to-Right Side Effects!

```
triangle draw
```

vs

```
(draw (triangle))
```

Stack Languages were all but forgotten during the 1990s

That’s Not Totally True

That’s Not Totally True

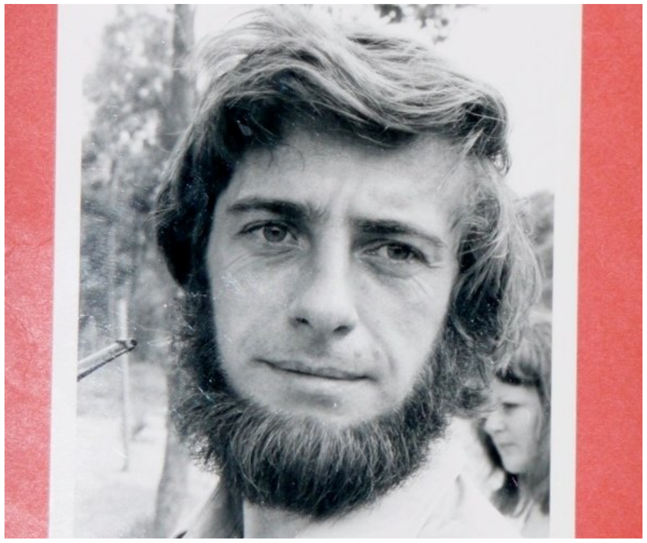
- Well Known Stack VMs
  - CPython Bytecode (1991)
  - Java Virtual Machine (1995)

That’s Not Totally True

- Well Known Stack VMs
  - CPython Bytecode (1991)
  - Java Virtual Machine (1995)
- Portability became important
  - Stack machines are easy to implement!

Joy
(2001)

A Purely Functional, Concatenative Language



Manfred von Thurn

Purely Functional

Every word can be thought of as a function of type **Stack → Stack**

Purely Functional

Every word can be thought of as a function of type **Stack** $\rightarrow$ **Stack**

Program Concatenation

==

Function Composition

Factor

(2003)

A modern, dynamic, practical stack-language

Slava Pestov

Factor : Forth :: Clojure : Lisp

```
! Programs as data: "Quotations"
[ 2 * 1 + ]  ! Pushes quotation onto stack
5 swap       ! stack:   5 [ 2 * 1 + ]
call         ! stack:  11
```

```
! Programs as data: "Quotations"
[ 2 * 1 + ]   ! Pushes quotation onto stack
5 swap        ! stack:    5 [ 2 * 1 + ]
call          ! stack:   11

! Higher order words: “Combinators”
{ 5 10 15 } [ 2 * ] map
! pushed { 10 20 30 }
! Programs as data: "Quotations"
[ 2 * 1 + ]   ! Pushes quotation onto stack
5 swap        ! stack:   5 [ 2 * 1 + ]
call          ! stack:  11

! Higher order words: “Combinators”
{ 5 10 15 } [ 2 * ] map
! pushed { 10 20 30 }

! 11 is still on the stack
[ 1 - ] dip   ! stack:  10 { 10 20 30 }
clear         ! empty stack
```

```
: double ( x -- y ) 2 * ;

: square ( x -- y ) dup * ;

: inc ( x -- y ) 1 + ;

: dec ( x -- y ) 1 - ;
: double ( x -- y ) 2 * ;

: square ( x -- y ) dup * ;

: inc ( x -- y ) 1 + ;

: dec ( x -- y ) 1 - ;

: plus-minus ( x -- y z )
    [ inc ] [ dec ] bi ;
```

```
: double ( x -- y ) 2 * ;

: square ( x -- y ) dup * ;

: inc ( x -- y ) 1 + ;

: dec ( x -- y ) 1 - ;

: plus-minus ( x -- y z )
    [ inc ] [ dec ] bi ;

5 10 15
[ double ] [ square ] [ plus-minus ]
tri*
! stack: 10 100 16 14
```

```
: print-zeroness ( n -- )
    0 = [
        "zero"
    ] [
        "non-zero"
    ] if print ;
```

```
: print-sign ( n -- )
    { { [ dup 0 > ] [ drop "positive" ] }
      { [ 0 < ] [ "negative" ] }
      [ "zero" ]
    } cond print ;
```

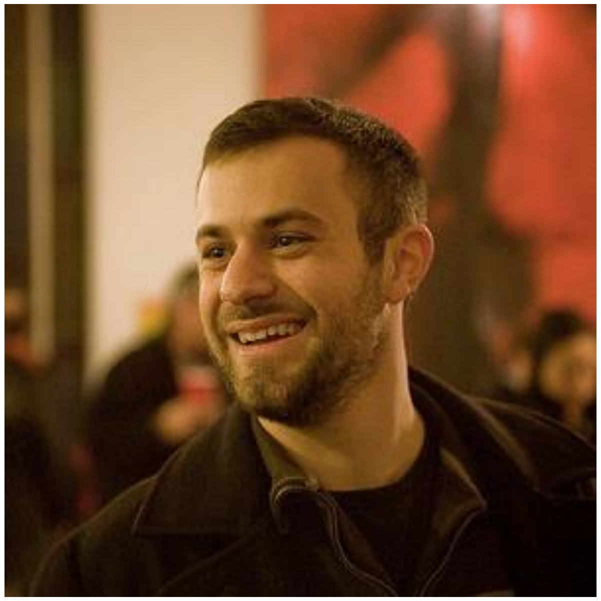
```
: print-sign ( n -- )
    sgn {
        {  1 [ "positive" ] }
        {  0 [ "zero" ] }
        { -1 [ "negative" ] }
    } case print
```

```
! Concatenation is Composition
10                               !  10
2 * 1 +                          !  21
[ 2 * 1 + ] call                 !  43
[ 2 * ] [ 1 + ] compose call     !  87
clear

! Prepending is "Right-Currying"
{ 5 10 15 } 2 [ - ] curry map ! { 3 8 13 }
```

Factor
Implementation
Utilities
protocol
words
classes
arrays
object
support
sequences
source
sequence
Lowlevel
parsing
reading
Data
Defining
introspection
generation
Arithmetic
native
byte
writing
Generic
compilation
method
constant
character
streams
Stack
word
Checksum
trings
Fixedsi
Grow
output
layout
UTF16
UTF8
image
String
buffers
files
units
structs
Directed
Complex
continuations
generalized
definitions
extensible
Singleton
description
Heap
Mixin
Definition
firstclass
CRC32
encoding
symbol
endian
crossreferencing
conditionals
syntax
queries
construction
Merge
dataflow
Builtin
bytes
numbers
group
cond
little
big
tupledefinition
system
Math
restorating
ASCII
library
array
catching
types
flow
effects
quotations
Common
directly
ANSI
capturing
destructors
fixedlength
Efficient
file
objects
Tuple
used
utility
errors
order
effect
time
elements
grouping
declarations
spread
control
cleave
Union
case
CPU
sort
vice
point
unions
barebones
interface
lowlevel
theoretic
tions
expect
os

Factjor
(2013)

A concatenative DSL for Clojure



Um, it's ME!

```clojure
(ns factjor.demo
  (:refer-clojure :only () :as clj)
  (:use factjor.core))
```

```clojure
(ns factjor.demo
  (:refer-clojure :only () :as clj)
  (:use factjor.core))

(run 5 2 * 1 +)             ;  (11)
```

```clojure
(ns factjor.demo
  (:refer-clojure :only () :as clj)
  (:use factjor.core))

(run 5 2 * 1 +)           ;  (11)

(run 7 [inc] [dec] bi)  ;  (6 8)
```

```
(ns factjor.demo
  (:refer-clojure :only () :as clj)
  (:use factjor.core))

(run 5 2 * 1 +)             ;  (11)

(run 7 [inc] [dec] bi)      ;  (6 8)
                       ;  NOTE stack order!
```

```
(ns factjor.demo
  (:require [factjor.core :as cat]))


(cat/run 5 2 cat/* 1 cat/+)        ;  (11)

(cat/run
  7 [cat/inc] [cat/dec] cat/bi)   ;  (6 8)
```

```clojure
(ns factjor.demo
  (:require [clojure.core :as app]
            [factjor.core :as cat]))
```

```
(ns factjor.demo
  (:require [clojure.core :as app]
            [factjor.core :as cat]))

(cat/run (app/* 5 2) 1 cat/+)  ;  (11)
```

```
;; Each of these evaluate to 11
(cat/run

  (app/* 5 2) 1 cat/+

  (cat/* 5 2) 1 cat/+

  (app/* 5 2) (cat/+ 1)

)
```

```
(def sixteen [6 10 cat/+])
```

```
(def sixteen [6 10 cat/+])

(cat/run sixteen cat/call)  ;  (16)
```

```
(def sixteen [6 10 cat/+])

(cat/run sixteen cat/call)  ;  (16)

(def square [cat/dup cat/*])
```

```
(def sixteen [6 10 cat/+])

(cat/run sixteen cat/call)  ;  (16)

(def square [cat/dup cat/*])

(def composed (concat sixteen square))
```

```
(def sixteen [6 10 cat/+])

(cat/run sixteen cat/call)  ;  (16)

(def square [cat/dup cat/*])

(def composed (concat sixteen square))

(apply cat/run composed)  ;  (256)
```

```clojure
(ns factjor.demo
  (:require [factjor.core :as cat
              :refer (defword)]))

(defword square [x -- y] cat/dup cat/*)

(cat/run 5 square)  ;  (25)
```

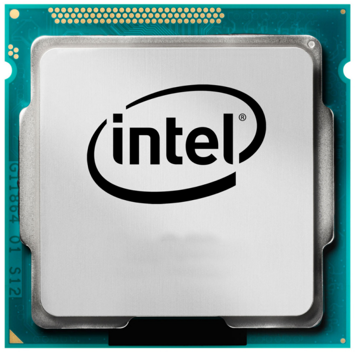
```
(ns factjor.demo
  (:require [factjor.core :as cat
             :refer (defword defprim)]))

(defprim divmod [x y -- q r]
  (conj $ (quot x y) (mod x y)))

(cat/run 5 2 divmod)  ;  (1 2)
```

DomScript

PostScript for the DOM

DomScript

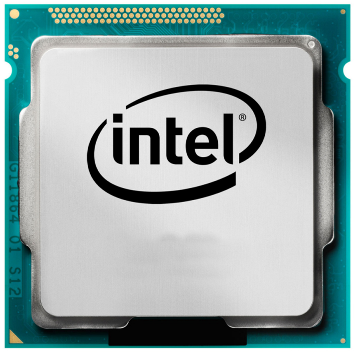
PostScript for the DOM



DomScript

PostScript for the DOM

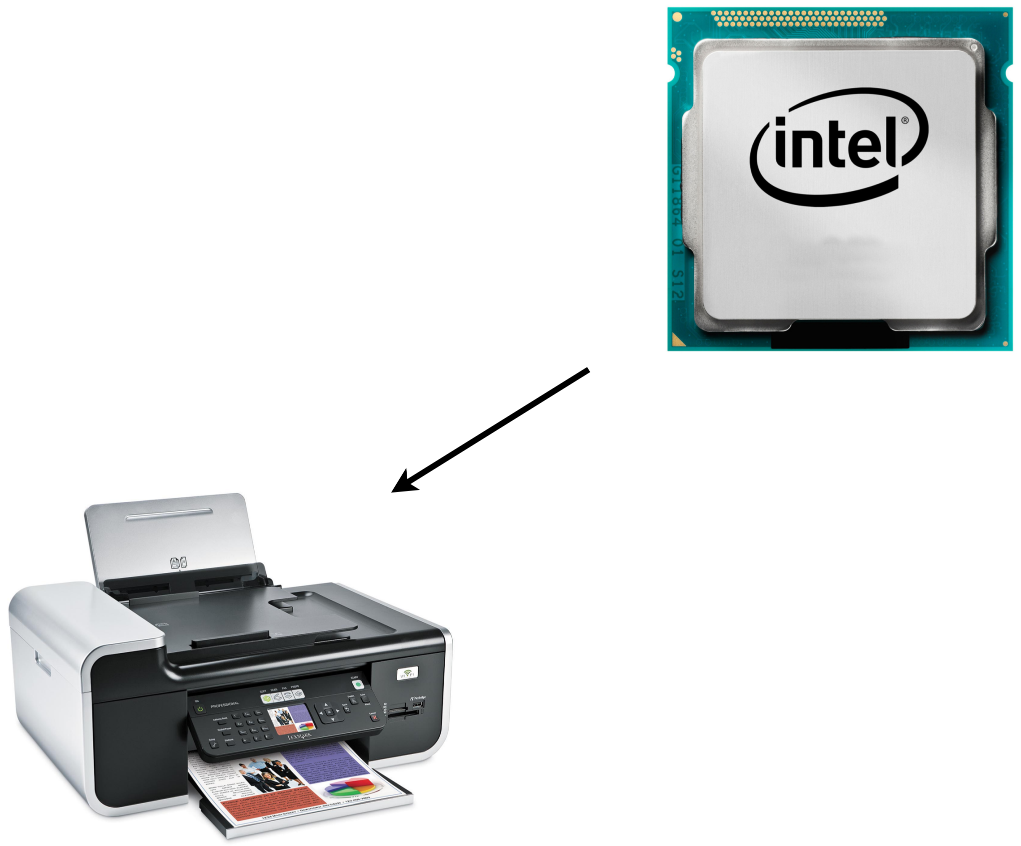
Key Idea: Procedures as Data!



DomScript

PostScript for the DOM

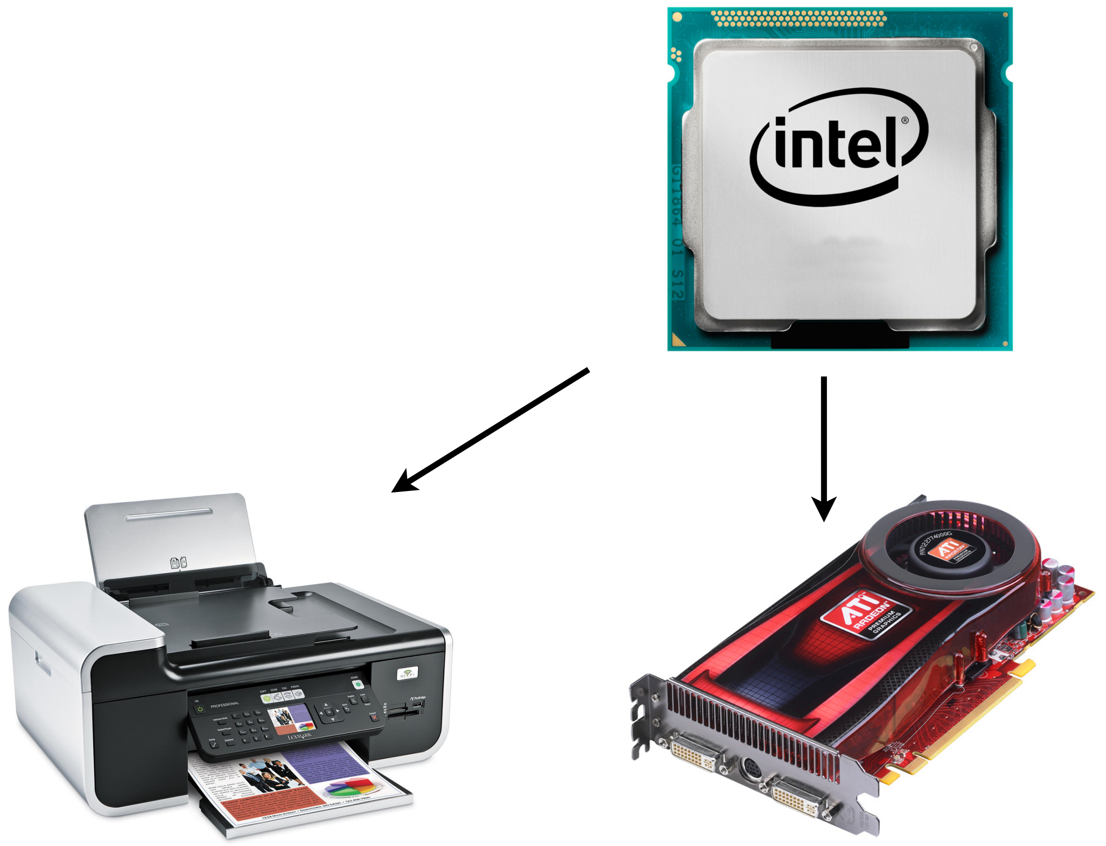
Key Idea: Procedures as Data!



DomScript

PostScript for the DOM

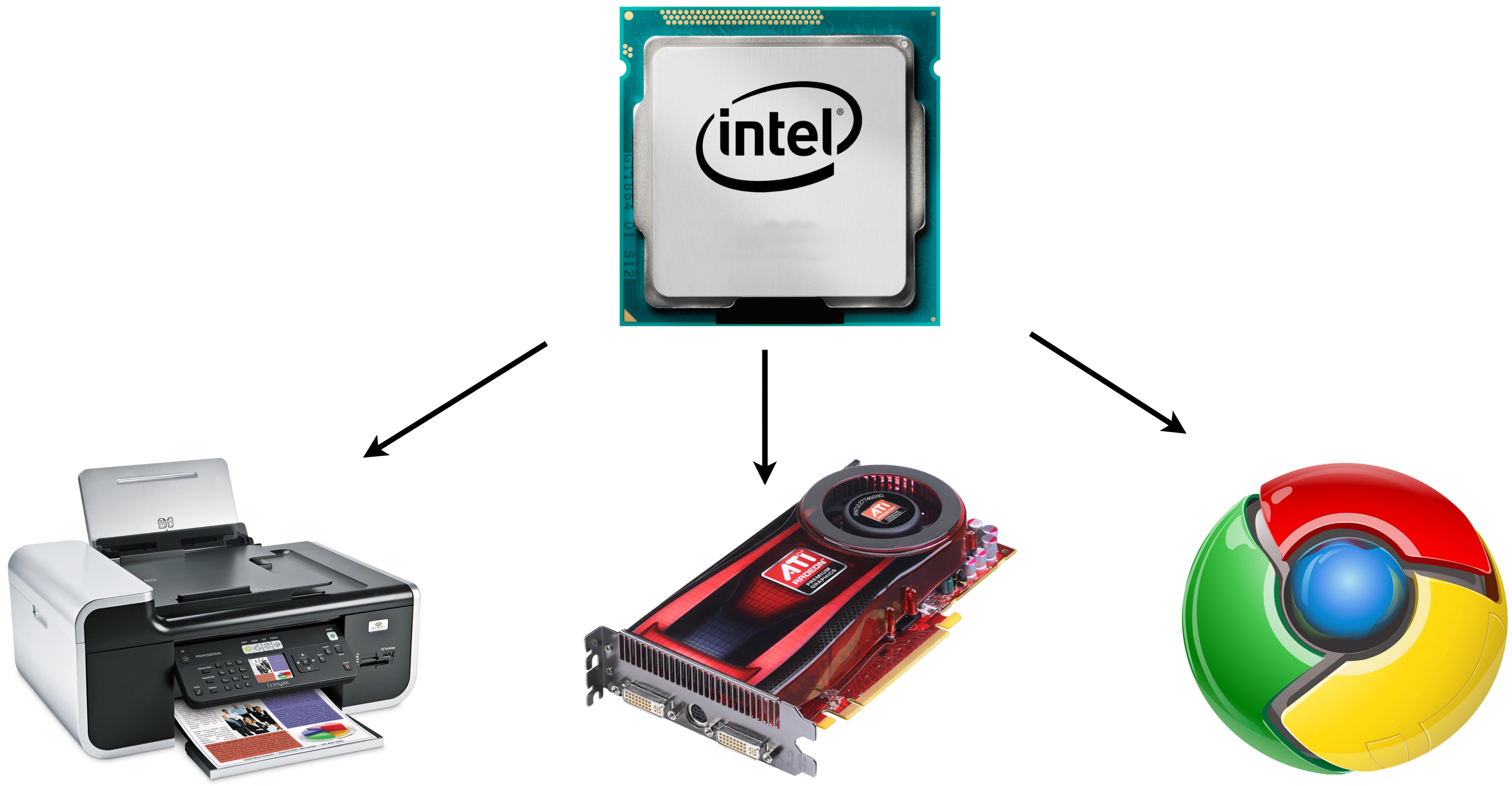
Key Idea: Procedures as Data!

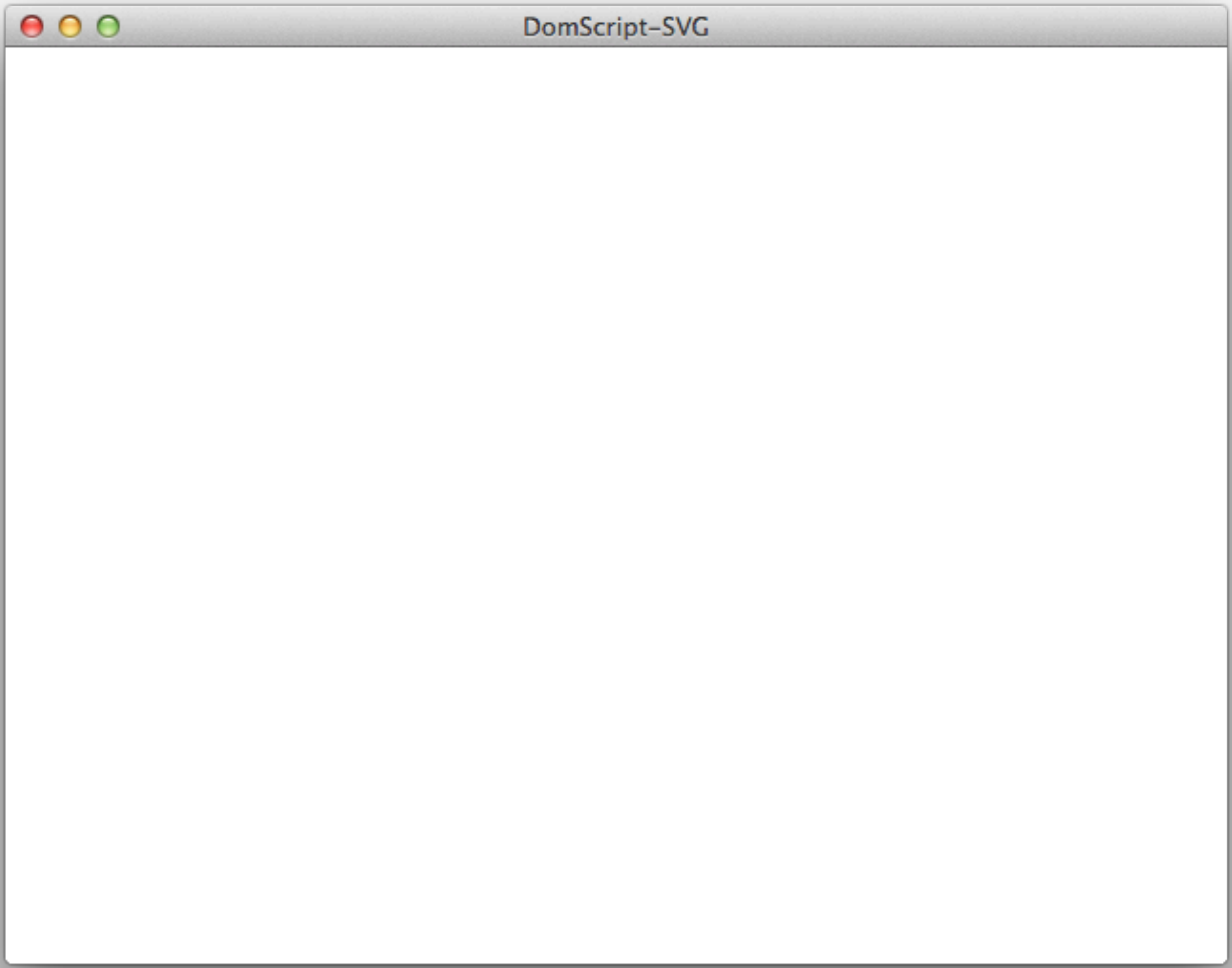


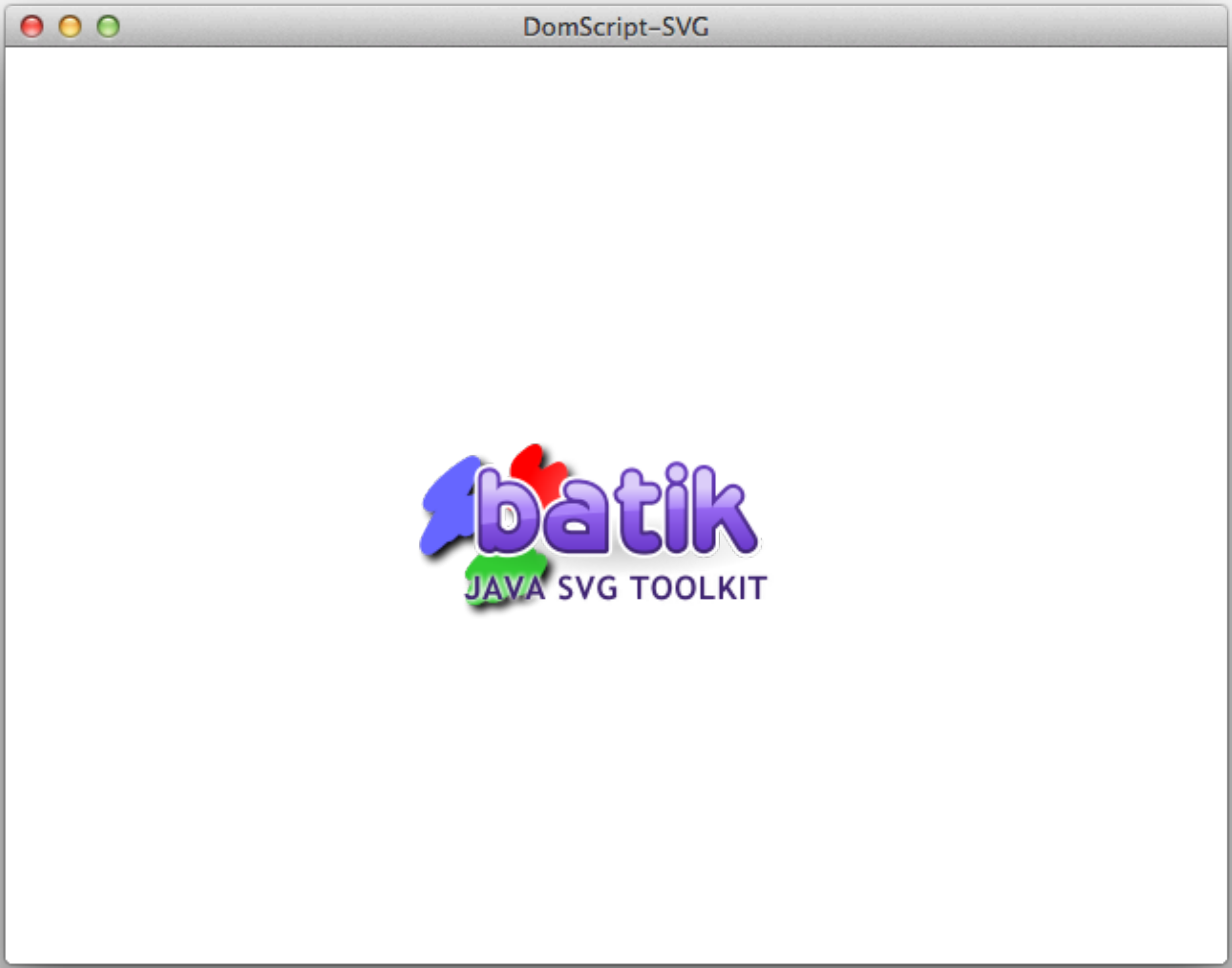
DomScript

PostScript for the DOM

Key Idea: Procedures as Data!



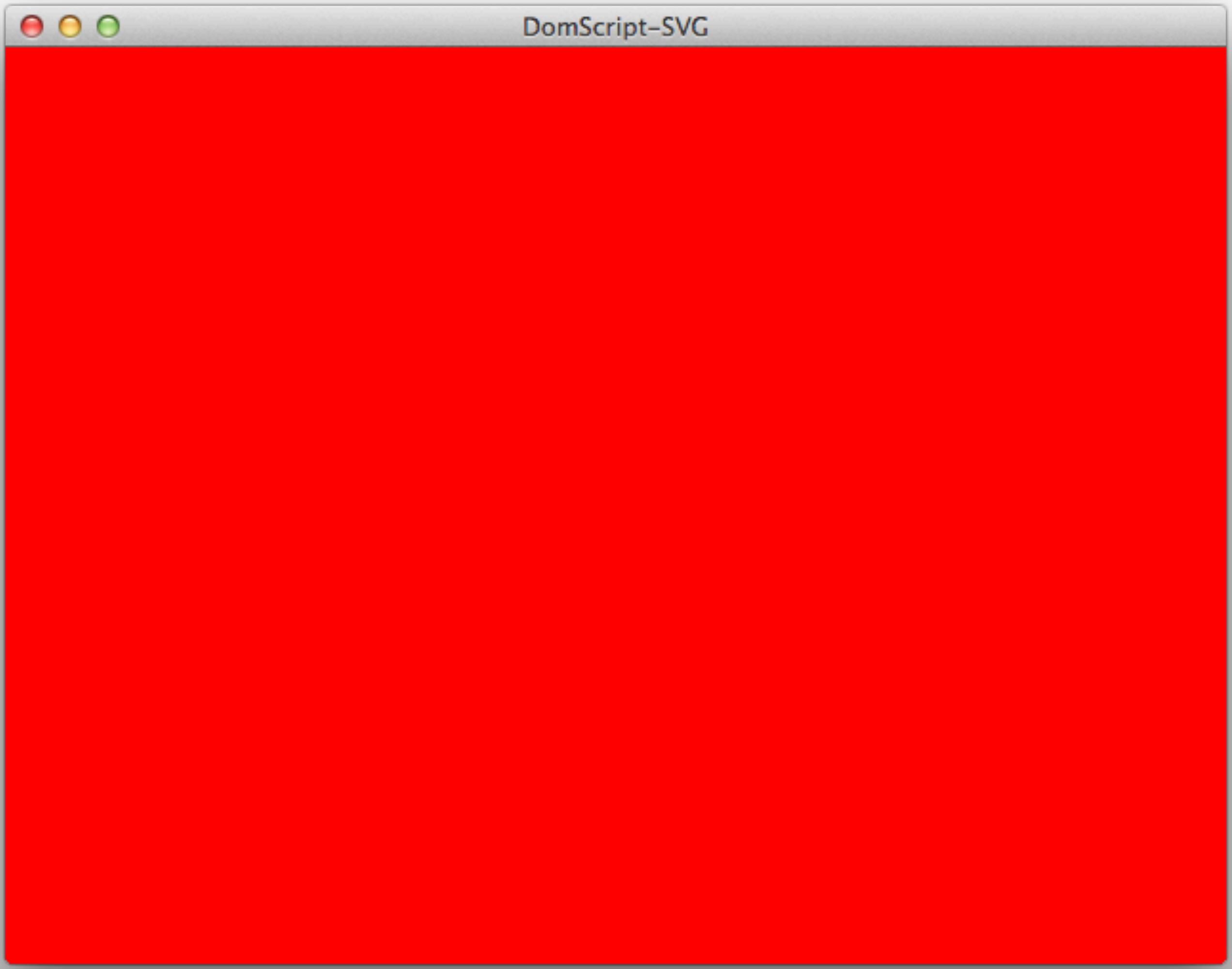

DomScript-SVG


DomScript-SVG
batik
JAVA SVG TOOLKIT

DomScript-SVG
batik
JAVA SVG TOOLKIT

```
(go
  document-element
  :svg/rect create-element
  :x 0 set-attribute
  :y 0 set-attribute
  :width 640 set-attribute
  :height 480 set-attribute
  :fill "red" set-attribute
  append
)
```

```clojure
(go
  document-element
  (create-element :svg/rect)
  (set-attribute :x 0)
  (set-attribute :y 0)
  (set-attribute :width 640)
  (set-attribute :height 480)
  (set-attribute :fill "red")
  append
)
```


DomScript-SVG

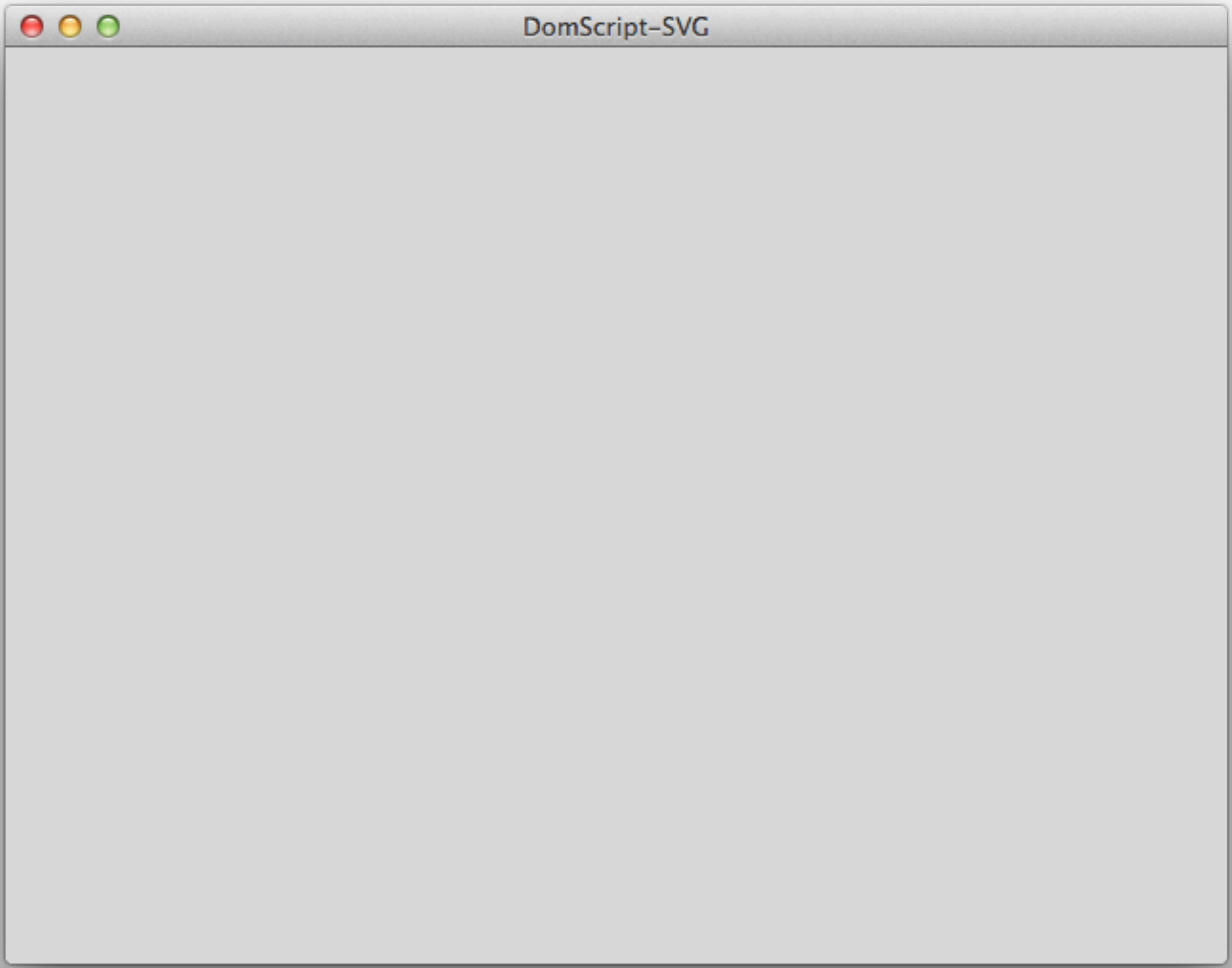
```
(go
  document-element
  (create-element :svg/rect)
  (set-attribute :x 0)
  (set-attribute :y 0)
  (set-attribute :width 640)
  (set-attribute :height 480)
  (set-attribute :fill "red")
  append
)
```

```
;; If DomScript were applicative...
(append
  (document-element)
  (-> (create-element :svg/rect)
      (set-attribute :x 0)
      (set-attribute :y 0)
      (set-attribute :width 640)
      (set-attribute :height 480)
      (set-attribute :fill "red")))
```

```clojure
;; Need locals to preserve execution order
(let [parent (document-element)
      child (-> (create-element :svg/rect)
                (set-attribute :x 0)
                (set-attribute :y 0)
                (set-attribute :width 640)
                (set-attribute :height 480)
                (set-attribute :fill "red"))]
  (append parent child))
```

```
(go
  document-element children
  (set-attributes {:fill "black"
                   :opacity 0.15})
)
```

```
(go
  document-element children cat/first
  (set-attributes {:fill "black"
                   :opacity 0.15})
)
```

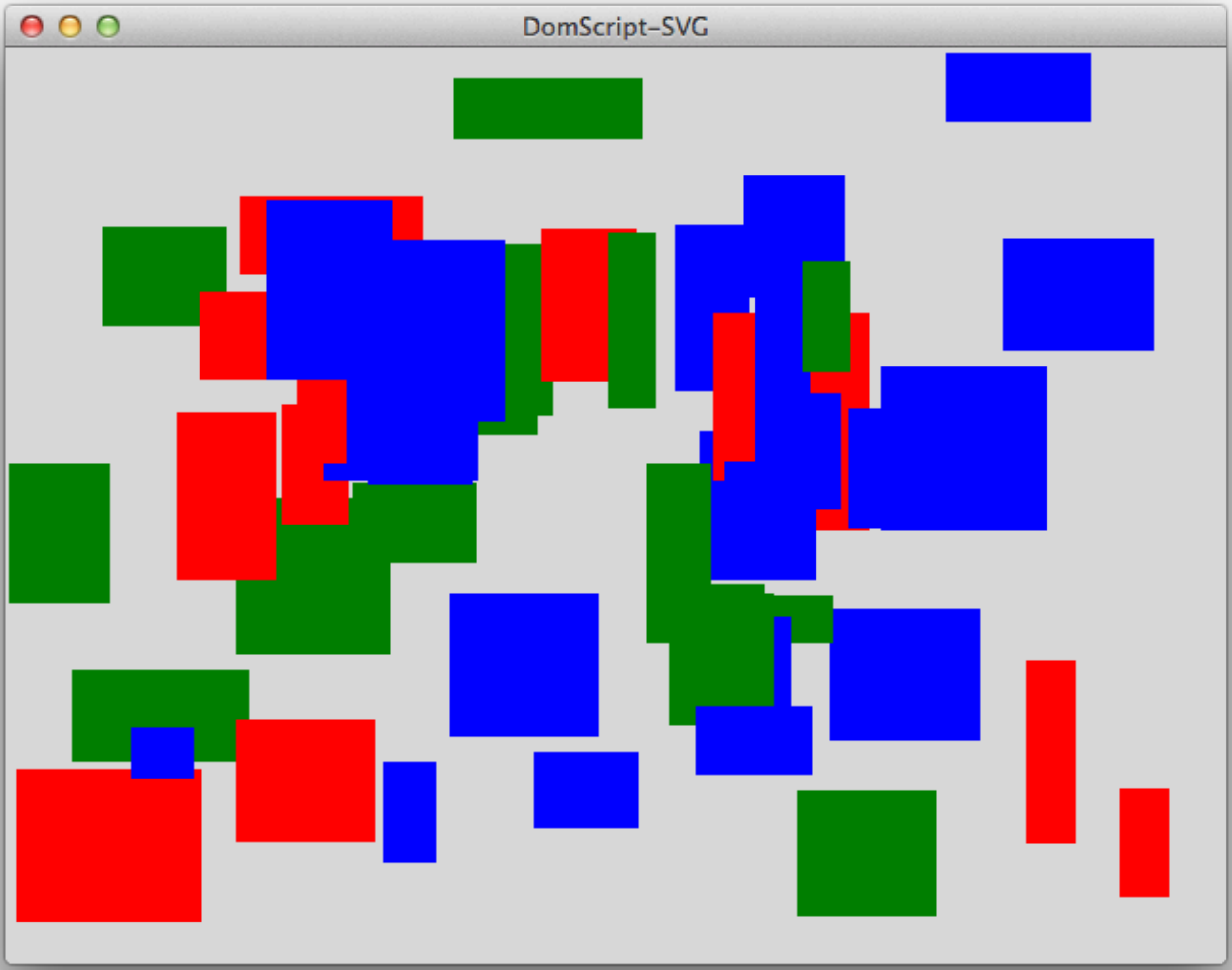

DomScript-SVG

```
(defn random-rect []
  (let [w (+ (rand-int 75) 25)
        h (+ (rand-int 75) 25)
        x (rand-int (- 640 w))
        y (rand-int (- 480 h))
        c (rand-nth ["red" "green" "blue"])]
    ...)
```

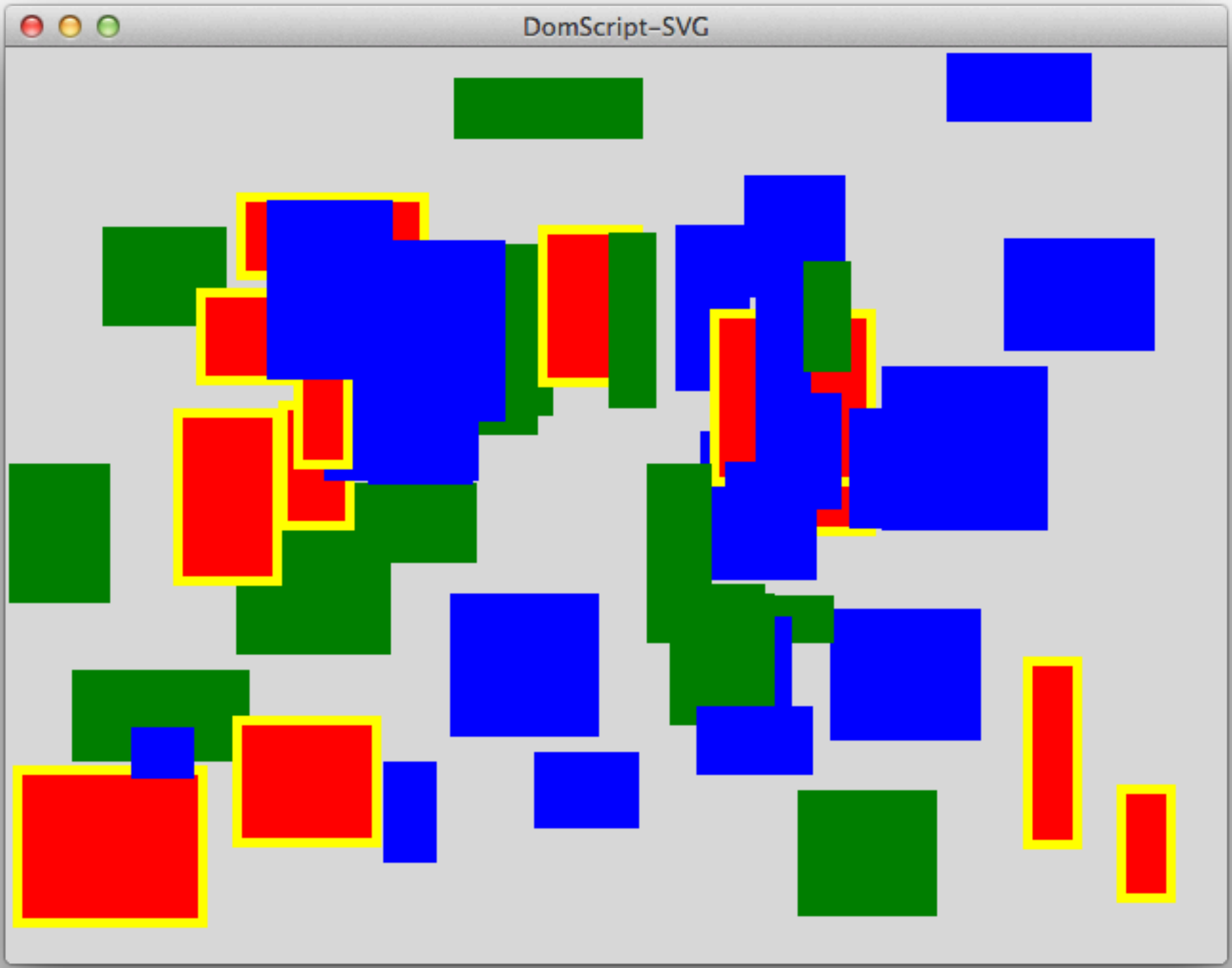
```
(defn random-rect []
  (let [w (+ (rand-int 75) 25)
        h (+ (rand-int 75) 25)
        x (rand-int (- 640 w))
        y (rand-int (- 480 h))
        c (rand-nth ["red" "green" "blue"])]
    ;; Returns code as data!
    [(create-element :svg/rect)
     (set-attributes {:x x :y y
                      :width w :height h
                      :fill c})
     append]))
```

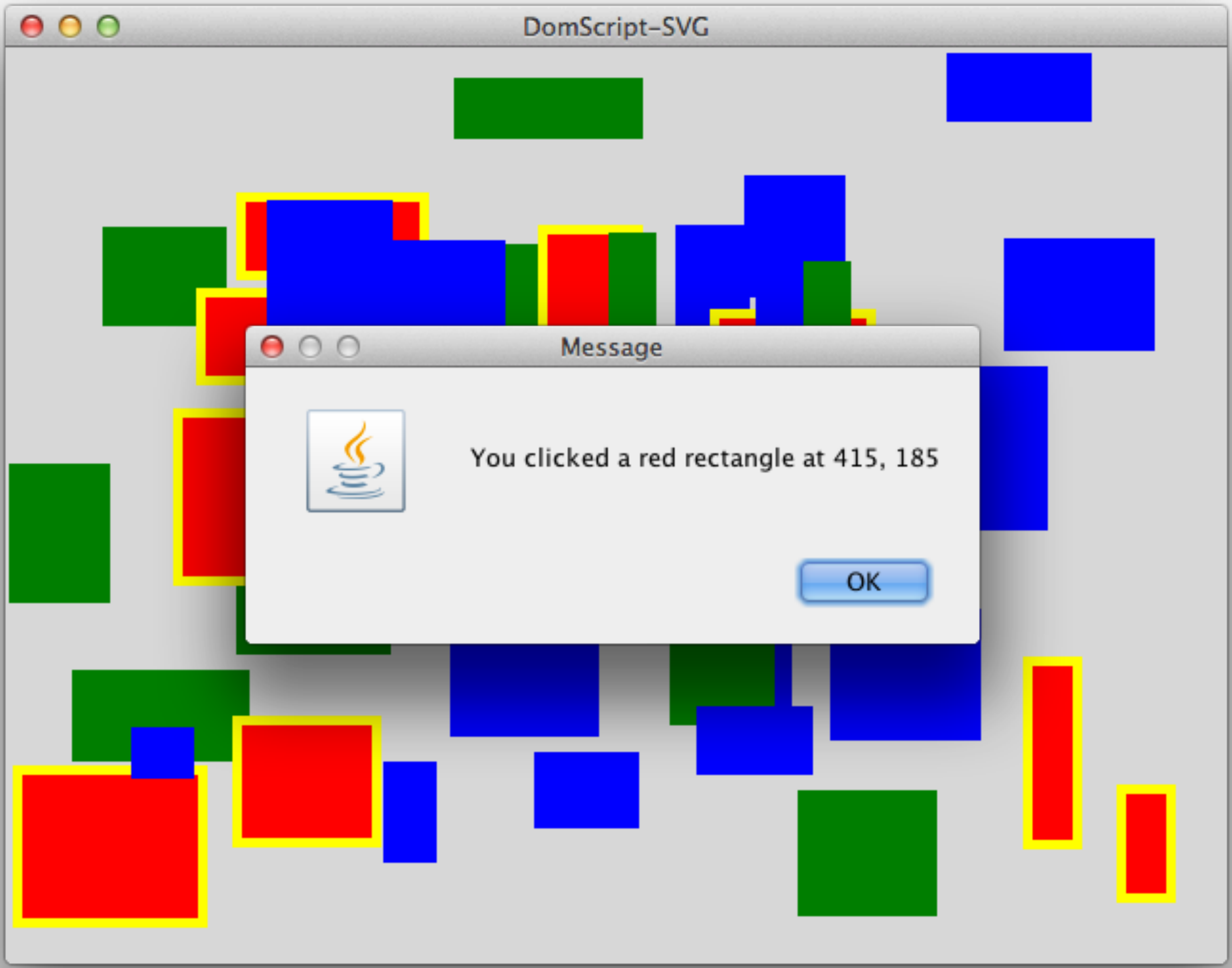
```clojure
(defn random-rect []
  (let [w (+ (rand-int 75) 25)
        h (+ (rand-int 75) 25)
        x (rand-int (- 640 w))
        y (rand-int (- 480 h))
        c (rand-nth ["red" "green" "blue"])]
    ;; Returns code as data!
    [(create-element :svg/rect)
     (set-attributes {:x x :y y
                      :width w :height h
                      :fill c})
     append]))

(random-rect) ; No external effect.
```

```
(apply go
  document-element
  (apply concat
    (repeatedly 50 random-rect)))
```


DomScript-SVG

```
(go
  (select "rect[fill=red]")
  (set-attribute :stroke-width 5)
  (set-attribute :stroke "yellow")
  (bind :click ::event-key
    (fn [event]
      (alert (str
        "You clicked a red rectangle at "
        (:x event) ", " (:y event))))))
)
```


DomScript-SVG



You clicked a red rectangle at 415, 185

	Applicative	Concatenative

	Applicative	**Concatenative**
Abstraction	Functions	Procedures

	Applicative	**Concatenative**
Abstraction	Functions	Procedures
Shape	Trees	Sequences

	Applicative	**Concatenative**
Abstraction	Functions	Procedures
Shape	Trees	Sequences
Nature	Declarative	Imperative

	Applicative	**Concatenative**
Abstraction	Functions	Procedures
Shape	Trees	Sequences
Nature	Declarative	Imperative
As Data	Retained Documents	Immediate Commands

	Applicative	**Concatenative**
Abstraction	Functions	Procedures
Shape	Trees	Sequences
Nature	Declarative	Imperative
As Data	Retained Documents	Immediate Commands
On The Wire	Batches	Streams

Concatenative Clojure

Brandon Bloom

brandon.d.bloom@gmail.com
github.com/brandonbloom
twitter.com/brandonbloom