Designing Macros
with syntax-parse

Ryan Culpepper
PLT, University of Utah

Q:

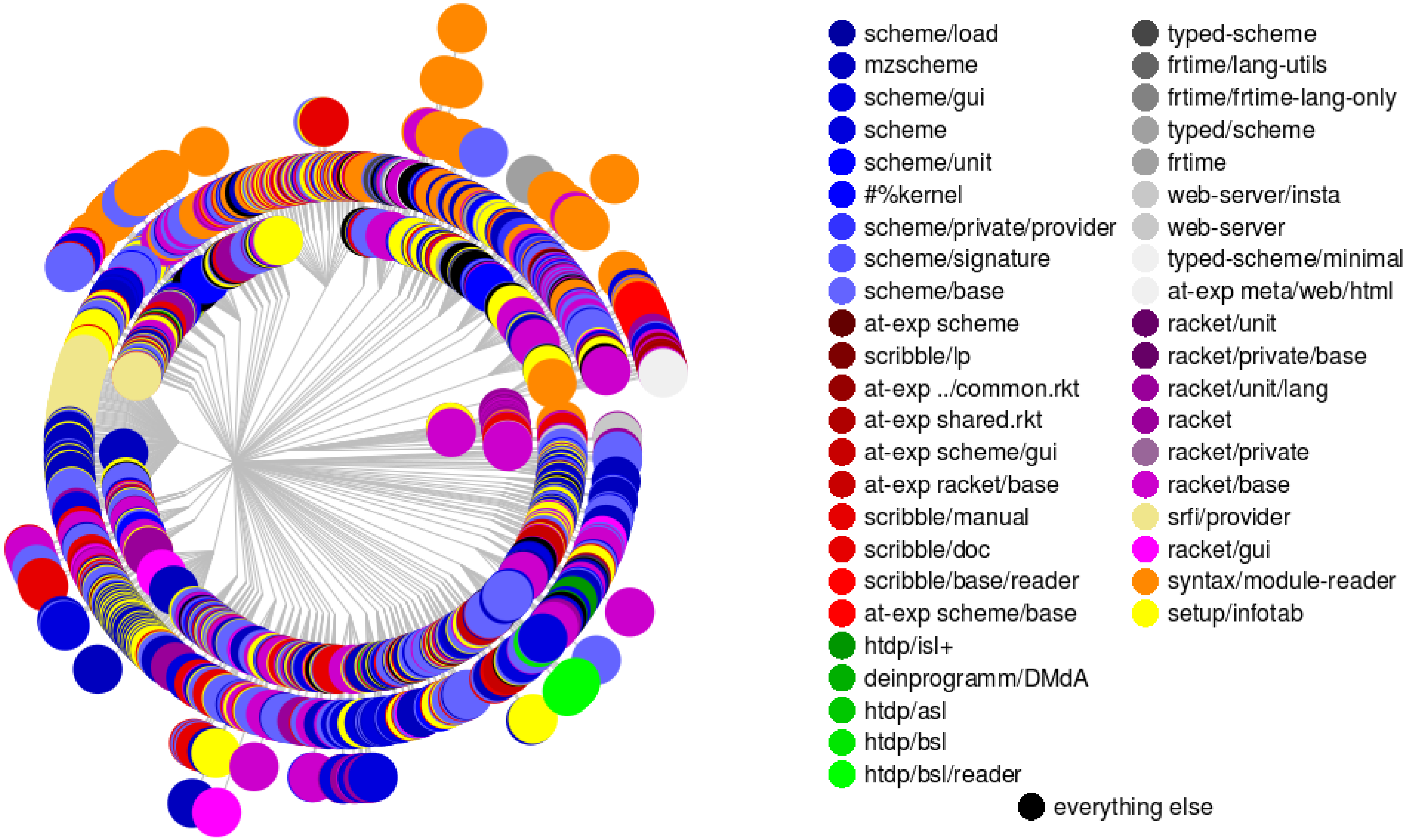
What are macros good for?

Q:

What are macros good for?

A (1):

Growing big languages from small ones and experimenting with language features


scheme/load
mzscheme
scheme/gui
scheme
scheme/unit
#%kernel
scheme/private/provider
scheme/signature
scheme/base
at-exp scheme
scribble/lp
at-exp ../common.rkt
at-exp shared.rkt
at-exp scheme/gui
at-exp racket/base
scribble/manual
scribble/doc
scribble/base/reader
at-exp scheme/base
htdp/isl+
deinprogramm/DMdA
htdp/asl
htdp/bsl
htdp/bsl/reader
typed-scheme
frtime/lang-utils
frtime/frtime-lang-only
typed/scheme
frtime
web-server/insta
web-server
typed-scheme/minimal
at-exp meta/web/html
racket/unit
racket/private/base
racket/unit/lang
racket
racket/private
racket/base
srfi/provider
racket/gui
syntax/module-reader
setup/infotab
everything else

Q:

What are macros good for?

A (2):

Creating lightweight domain-specific languages for new and existing domains

Slideshow

picts and slides

Findler and Flatt, JFP 2006

Slideshow

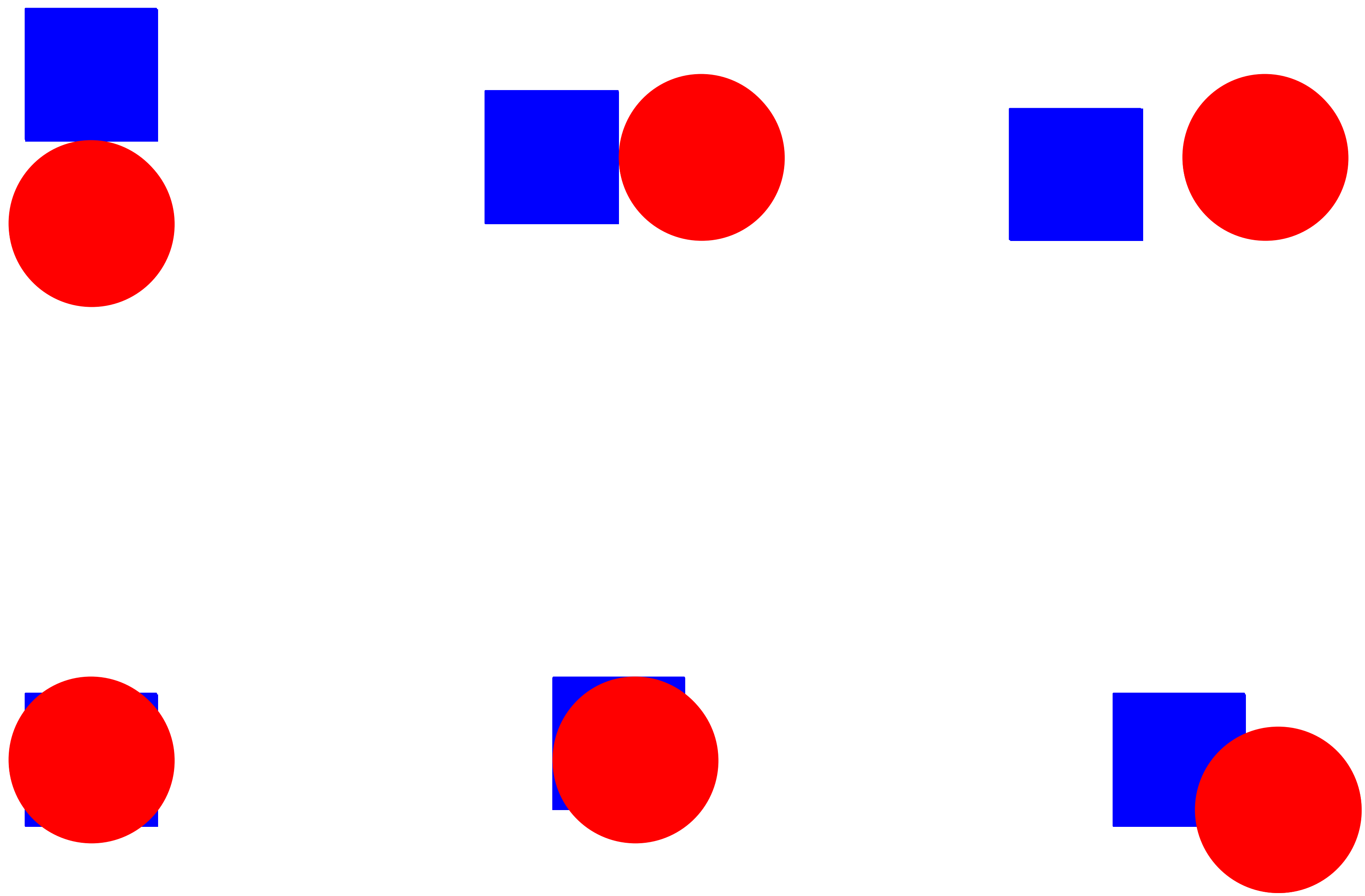
picts and slides

killer app!

Findler and Flatt, JFP 2006

text

```
(define (fact n)
  (if (zero? n)
      1
      (* n (fact (sub1 n)))))
```



```
(define (fact n)
  (if (zero? n)
      1
      (* n (fact n))))
```

text

text

```
(define (fact n)
  (if (zero? n)
      1
      (* n (fact n))))
```

title

paragraphs

- and items

and other things

vertically appended

title

paragraphs

- and items

and other things

vertically appended

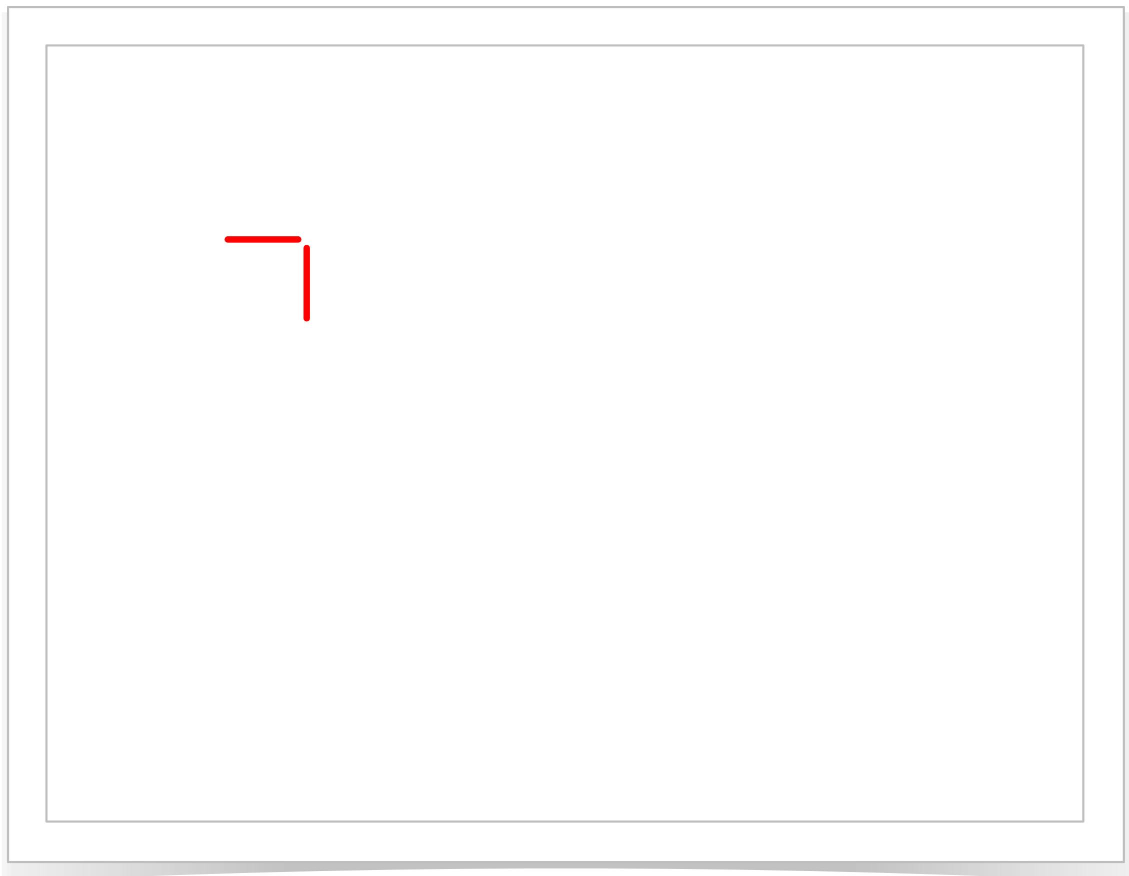
(also: transitions, alternatives, animation) pict arithmetic and slide model

vs.

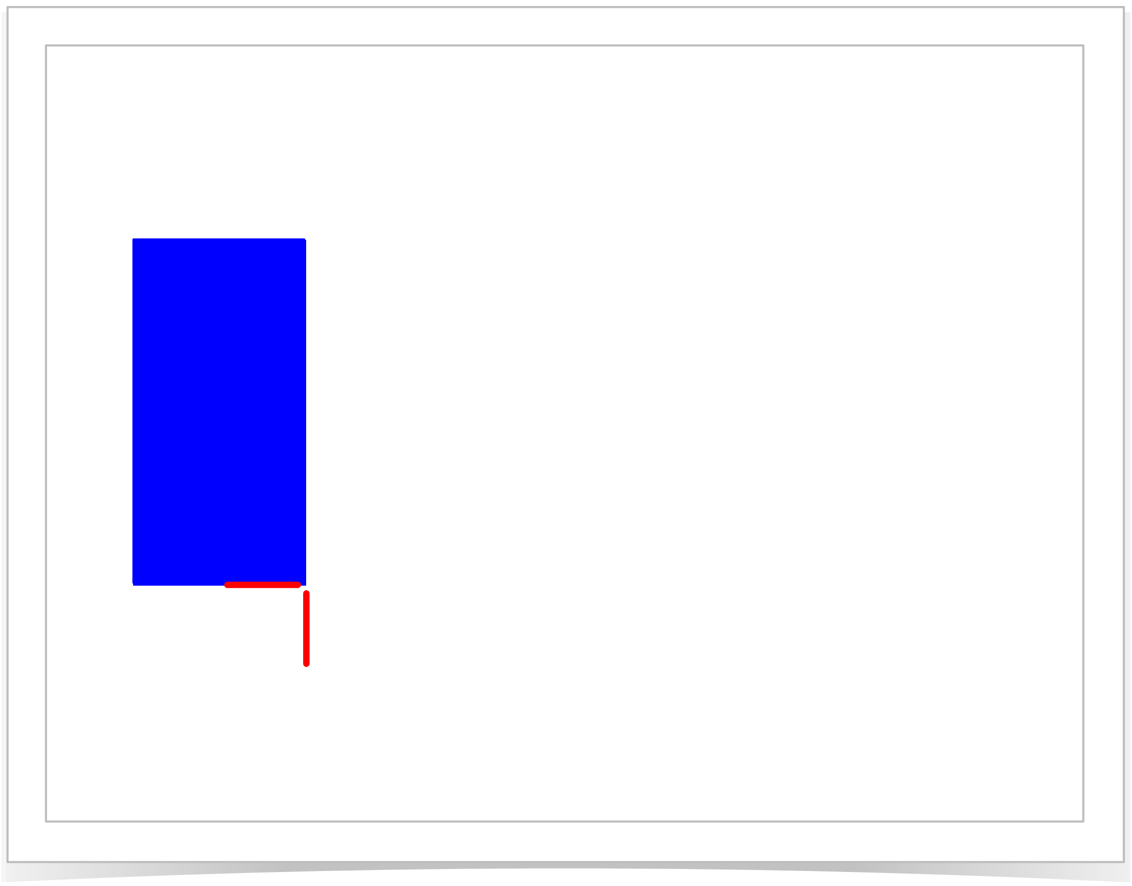
canvas and cursor model

```
(ppict-do
 (empty-slide))
```

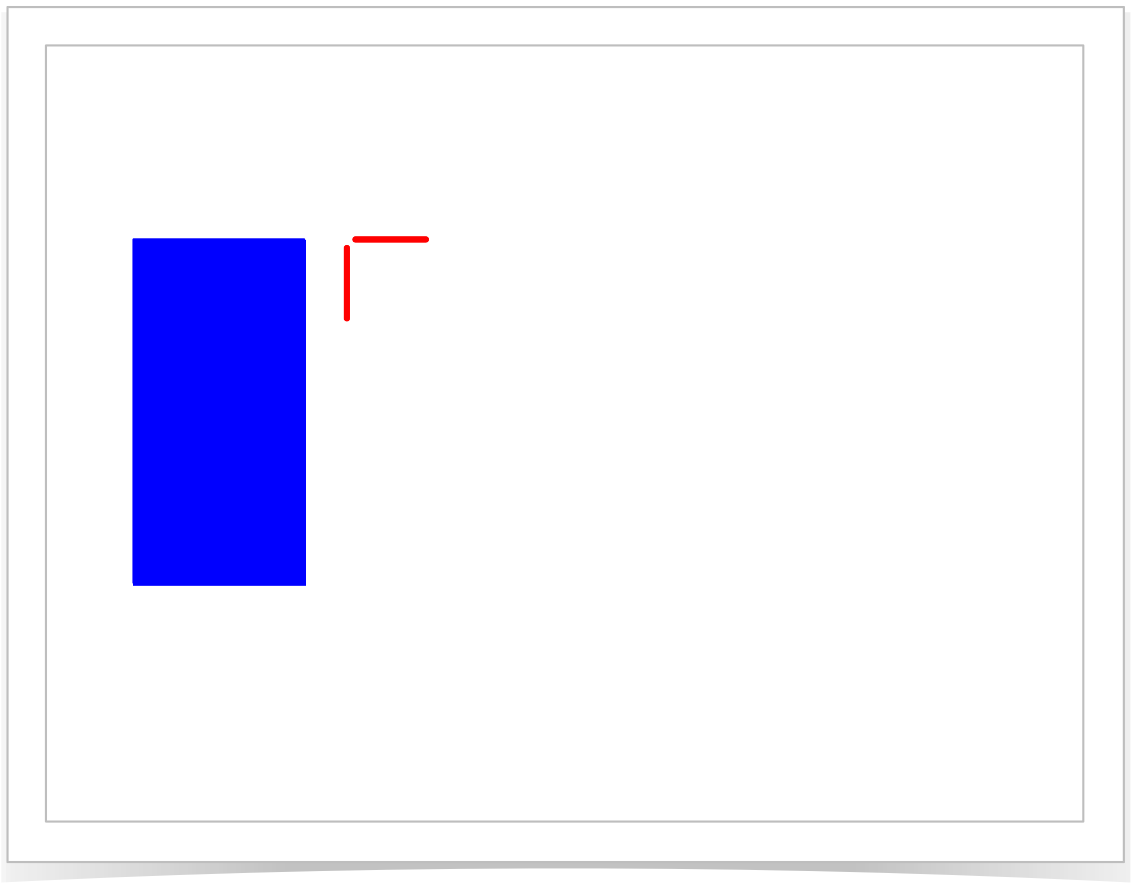
```
(ppict-do
 (empty-slide)
 #:go (coord 1/4 1/4 'rt))
```



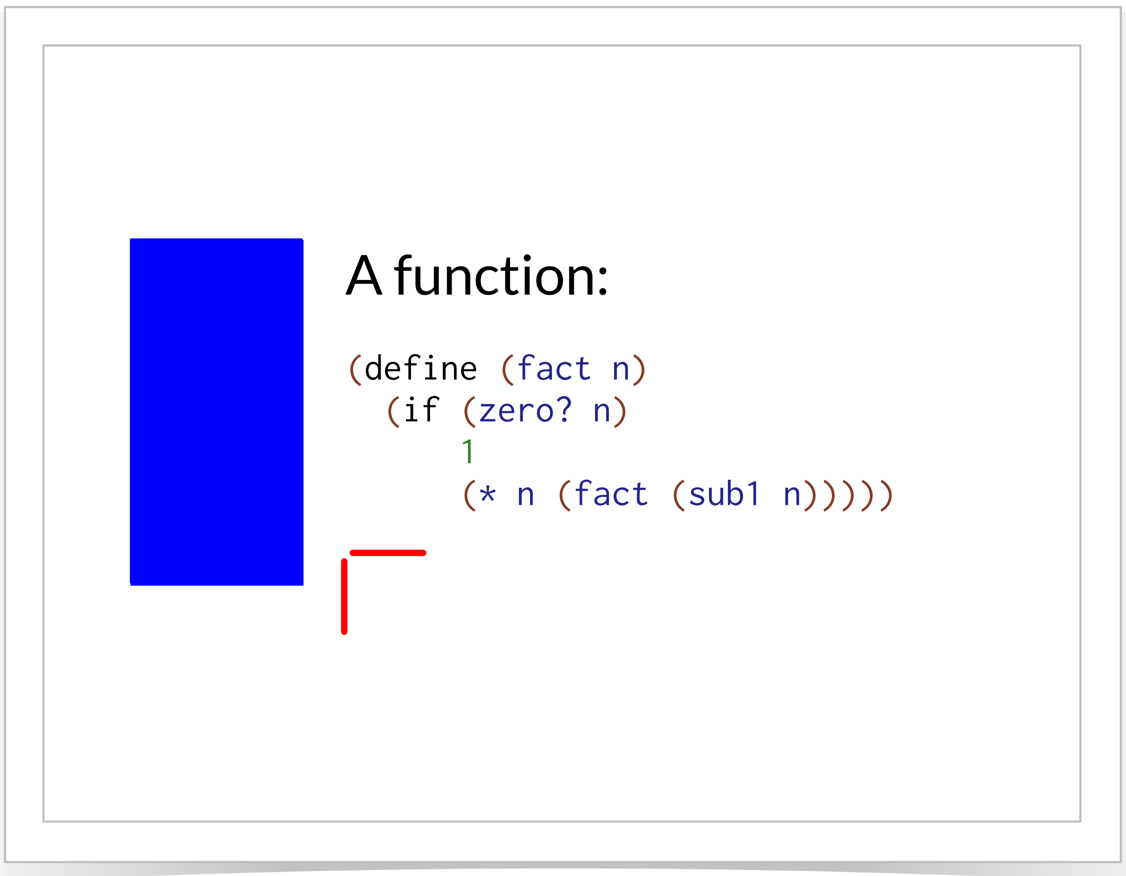
```
(ppict-do
 (empty-slide)
 #:go (coord 1/4 1/4 'rt)
 (colorize (filled-rectangle 100 200)
           "blue"))
```



```
(ppict-do
 (empty-slide)
 #:go (coord 1/4 1/4 'rt)
 (colorize (filled-rectangle 100 200)
           "blue")
 #:go (coord 1/4 1/4 'lt
             #:abs-x gap-size))
```



```
(ppict-do
 (empty-slide)
 #:go (coord 1/4 1/4 'rt)
 (colorize (filled-rectangle 100 200)
           "blue")
 #:go (coord 1/4 1/4 'lt
             #:abs-x gap-size)
 gap-size
 (t "A function:")
 (small
  (code (define (fact n)
          (if (zero? n)
              1
              (* n (fact (sub1 n)))))))
```



```
(ppict-do
 (empty-slide)
 #:go (coord 1/4 1/4 'rt)
 (colorize (filled-rectangle 100 200)
           "blue")
 #:go (coord 1/4 1/4 'lt
             #:abs-x gap-size)
 gap-size
 (t "A function:")
 (small
  (code (define (fact n)
          (if (zero? n)
              1
              (* n (fact (sub1 n))))))))
```

base pict

```
(ppict-do
 (empty-slide)
 #:go (coord 1/4 1/4 'rt)
 (colorize (filled-rectangle 100 200)
           "blue")
 #:go (coord 1/4 1/4 'lt
             #:abs-x gap-size)
 gap-size
 (t "A function:")
 (small
  (code (define (fact n)
          (if (zero? n)
              1
              (* n (fact (sub1 n))))))))
```

base pict

placer

```
(ppict-do
 (empty-slide)
 #:go (coord 1/4 1/4 'rt)
 (colorize (filled-rectangle 100 200)
           "blue")
 #:go (coord 1/4 1/4 'lt
             #:abs-x gap-size)
 gap-size
 (t "A function:")
 (small
  (code (define (fact n)
          (if (zero? n)
              1
              (* n (fact (sub1 n)))))))
```

```
(ppict-do
 (empty-slide)
 #:go (coord 1/4 1/4 'rt)
 (colorize (filled-rectangle 100 200)
           "blue")
 #:go (coord 1/4 1/4 'lt
             #:abs-x gap-size)
 gap-size
 (t "A function:")
 (small
  (code (define (fact n)
          (if (zero? n)
              1
              (* n (fact (sub1 n))))))))
```

base pict

placer

pict element

```
(ppict-do
 (empty-slide)
 #:go (coord 1/4 1/4 'rt)
 (colorize (filled-rectangle 100 200)
           "blue")
 #:go (coord 1/4 1/4 'lt
             #:abs-x gap-size)
 gap-size
 (t "A functi
 (small
  (code (define (fact n)
          (if (zero? n)
              1
              (* n (fact (sub1 n))))))))
```

base pict

placer

pict element

spacer element

```
(ppict-do base-expr fragment ...)

(pslide fragment ...)

fragment = elem-expr ...+
         | #:go placer-expr
         | #:next

elem-expr : (or/c pict? real?)

placer-expr : placer?
```

```
ppict-do-state : pict?
```

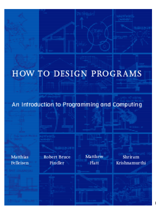
Q:

How do we design a macro like `ppict-do`?

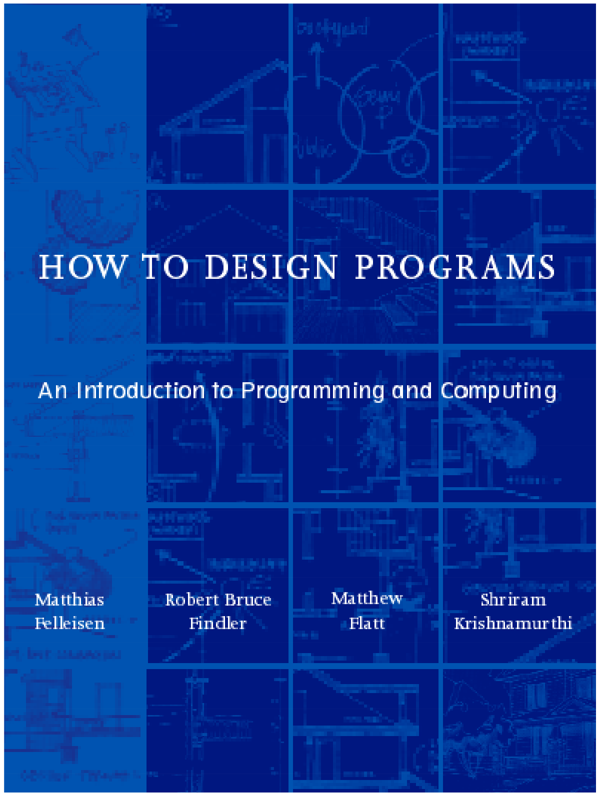
Q:

How do we design a macro like `ppict-do`?

A:



... roughly



Strategies

Atomic

Structured

Enumerated

Recursive

Part I: Atomic Data

expressions

identifiers

data to be quoted

```
expr = ???

expanded-expr = (#%plain-lambda formals expanded-expr)
              | (#%plain-app expanded-expr expanded-expr ...)
              | (if expanded-expr expanded-expr expanded-expr ...)
              | …
```

```
expr = ???

expanded-expr = (#%plain-lambda formals expanded-expr)
              | (#%plain-app expanded-expr expanded-expr ...)
              | (if expanded-expr expanded-expr expanded-expr ...)
```

local-expand

An expression:

- computes a value (or values)
- performs side effects
- depends on dynamic and static context

delay its execution

```
   (delay e)
➡ (make-promise (lambda () e))
```

delay its execution

```
  (delay e)
➡ (make-promise (lambda () e))
```

```
(define-syntax (delay stx)
  (syntax-parse stx
    [(_ e)
     #:declare e expr
     #'(make-promise (lambda () e))]))
```

change order or number of times executed

```
   (forever e)
➡ (let loop () (begin e (loop)))
```

change parameterization, exception handler, etc

```
  (without-output e)
➡ (parameterize ((current-output
                  (open-output-nowhere)))
    e)
```

change state before and after

```
   (with-lock e)

➡ (dynamic-wind
     (lambda () (semaphore-wait the-lock))
     (lambda () e)
     (lambda () (semaphore-post the-lock)))
```

change *static* context

```
   (ppict-do/1 base pict-elem)
➡ (let ([b base])
     (ppict-add b
                (with-ppict-do-state b
                  pict-elem)))

(define-syntax-rule (with-ppict-do-state b e)
  (syntax-parameterize
      ((ppict-do-state
        (make-rename-transformer #'b)))
    (list e)))
```

place a contract on its value

```
(define-syntax (ppict-do/2 stx)
  (syntax-parse stx
    [(_ base pict)
     #:declare base (expr/c #'ppict?
                            #:name "base argument")
     #:declare pict (expr/c #'pict?
                            #:name "pict element")
     #'(let* ([b base.c]
              [p (with-ppict-do-state b pict.c)])
         (ppict-add b p))]))
```

```
> (ppict-do/2 a-ppict (circle 10))
#<pict>
```

```
> (ppict-do/2 a-ppict "abc")
```

pict element of ppict-do/2: self-contract violation, expected pict?, given "abc" ...

Part II:
Structured Data

single terms

multiple terms

```
(my-let/1 binding body-expr)

binding = (var-id rhs-expr)
```

```
(define-syntax-class binding
  #:description "binding pair"
  #:attributes (var rhs)
  (pattern (var rhs)
           #:declare var identifier
           #:declare rhs expr))
```

```
(define-syntax (my-let/1 stx)

  (define-syntax-class binding
    #:description "binding pair"
    #:attributes (var rhs)
    (pattern (var rhs)
             #:declare var identifier
             #:declare rhs expr))

  (syntax-parse stx
    [(_ b body)
     #:declare b binding
     #:declare body expr
     #'((lambda (b.var) body) b.rhs)]))
```

```
(define-syntax (my-let/1 stx)

  (define-syntax-class binding
    #:description "binding pair"
    #:attributes (var rhs)
    (pattern (var:id rhs:expr)))

  (syntax-parse stx
    [(_ b:binding body:expr)
     #'((lambda (b.var) body) b.rhs)]))
```

```
> (my-let/1 (x 1) (+ x x))
2
```

```
> (my-let/1 x 1 (+ x x))
```

my-let/1: expected binding pair at: `x`

```
(ppict-do/3 base-expr go-fragment pict-expr)

go-fragment = #:go placer-expr

placer-expr : placer?
```

```
(define-splicing-syntax-class go-fragment
  #:description "#:go fragment"
  #:attributes (placer.c)
  (pattern (~seq #:go placer)
           #:declare placer
                     (expr/c #'placer?
                             #:name "placer argument")))
```

```
(define-syntax (ppict-do/3 stx)

  (define-splicing-syntax-class go-fragment
    #:description "#:go fragment"
    #:attributes (placer.c)
    (pattern (~seq #:go placer)
             #:declare placer
                       (expr/c #'placer?
                               #:name "placer argument")))

  (syntax-parse stx
    [(_ base go pict)
     #:declare base (expr/c #'pict?)
     #:declare go go-fragment
     #:declare pict (expr/c #'pict?)
     #'(ppict-add (ppict-go base go.placer.c)
                  (list pict.c))]))
```

Part II.5: Uniform Sequences

```
(ppict-do/4 base-expr go-fragment pict-expr ...+)

go-fragment = #:go placer-expr

placer-expr : placer?
```

```
(define-syntax (ppict-do/3 stx)

  (define-splicing-syntax-class go-fragment ____)

  (syntax-parse stx
    [(_ base go pict ...+)
     #:declare base (expr/c #'pict?)
     #:declare go go-fragment
     #:declare pict (expr/c #'pict?)
     #'(ppict-add (ppict-go base go.placer.c)
                  (list pict.c ...))]))
```

Part III: Enumerated Data & Recursion

One syntax class per "kind" of syntax

```
(ppict-do base-expr fragment ...)

fragment = elem-expr ...+
         | #:go placer-expr
         | #:next
```

```
(ppict-do base-expr fragment ...)

fragment = #:go placer-expr
         | elem/next ...+

elem/next = #:next
          | elem-expr
```

```
(ppict-do base-expr fragment ...)

fragment = #:go placer-expr
         | elem/next ...+

elem/next = #:next
          | elem-expr
```

```
(define-syntax-class element
  #:description "element"
  (pattern e
           #:declare e (expr/c #'(or/c pict? real?))
           #:with code #'e.c)
  (pattern #:next
           #:with code #''next))
```

```
(ppict-do base-expr fragment ...)

fragment = #:go placer-expr
         | elem/next ...+

elem/next = #:next
          | elem-expr
```

```
(define-syntax-class fragment-sequence
  (pattern (p ...+ . fs)
           #:declare p elem
           #:declare fs fragment-sequence
           #:with code
                  #'(lambda (base)
                      (fs.code (ppict-add base (list p.code ...)))))
  (pattern (g . fs)
           #:declare g go-sequence
           #:declare fs fragment-sequence
           #:with code
                  #'(lambda (base)
                      (fs.code (ppict-go base g.placer.c))))
  (pattern ()
           #:with code #'(lambda (base) base)))
```

The end