Teach Scheme!
A Checkpoint

Matthias Felleisen (PLT)
Northeastern University
Teach Scheme, NOT A Checkpoint

Matthias Felleisen (PLT)
Northeastern University
If functional programming is that good, why is it so difficult to learn?

TeachScheme!

Shriram Krishnamurthi

Matthew Flatt

Robby Findler

Kathi Fisler

Tuesday, September 28, 2010
If functional programming is that good, why is it so difficult to learn?

Shriram Krishnamurthi
Matthew Flatt
Robby Findler
Kathi Fisler

TeachScheme!
A functional programming language is typed, lazy, *and* pure.
A functional programming language is typed, lazy, and pure.
A functional programming language is typed, lazy, and pure.

Functional programming is one or many of:

typed

lazy

pure
A functional programming language is typed, lazy, and pure.

Functional programming is one or many of:
- typed
- lazy
- pure
- untyped
- strict
- impure
A functional programming language is typed, lazy, and pure.

Functional programming is one or many of:

- typed
- lazy
- pure
- untyped
- strict
- impure
- sequential
- parallel
- concurrent

Thank You, LFP/ICFP
POPL ’02: Errors Matter

ICFP ’09: Functional I/O
From FP to Design and Beyond
Functional Programming is Good (TM).
Functional Programming is Good (TM).
Functional Programming is Good (TM).

Design for the Working SE (in College)
Functional Programming is Good (TM).

Design for the Working SE (in College)

systematic procedure = how to design
```
(define (eval-> b)
  (cond
   [(eq? 't b) 't]
   [(eq? 'f b) 'f]
   [else (if (symbol=? (eval-> (first b)) 'f)
            (eval-> (third b))
            't)])
)

(check-equal? (eval-> ex1) 't)
(check-equal? (eval-> ex2) 'f)
```
(define (eval-> b)
  (cond
   [(eq? 't b) 't]
   [(eq? 'f b) 'f]
   [else (if (symbol=? (eval-> (first b)) 'f)
             (eval-> (third b))
             't)]))

(check-equal? (eval-> ex1) 't)
(check-equal? (eval-> ex2) 'f)
(define (eval-> b)
  (cond
   [(eq? 't b) 't]
   [(eq? 'f b) 'f]
   [else (if (symbol=? (eval-> (first b)) 'f)
           (eval-> (third b))
           't)])
  (check-equal? (eval-> ex1) 't)
  (check-equal? (eval-> ex2) 'f)
(define (eval-> b)
  (cond
   [(eq? 't b) 't]
   [(eq? 'f b) 'f]
   [else (if (symbol=? (eval-> (first b)) 'f)
            (eval-> (third b))
            't)]))

(check-equal? (eval-> ex1) 't)
(check-equal? (eval-> ex2) 'f)
(define (eval-> b)
  (cond
   [(eq? 't b) 't]
   [(eq? 'f b) 'f]
   [else (if (symbol=? (eval-> (first b)) 'f)
             (eval-> (third b))
             't)]
  ))

(check-equal? (eval-> ex1) 't)
(check-equal? (eval-> ex2) 'f)
(define (eval-> b)
  (cond
   [(eq? 't b) 't]
   [(eq? 'f b) 'f]
   [else (if (symbol=? (eval-> (first b)) 'f)
             (eval-> (third b))
             't)]))

(check-equal? (eval-> ex1) 't)
(check-equal? (eval-> ex2) 'f)

Design a program that does something like the program I did in class. Actually do ten or twenty or thirty of them,
Teaching by example doesn’t really work.
Design a program that does something like the program I did in class. Actually do ten or twenty or thirty of them.
Teaching features and tools doesn’t work either.
But functional programming works, right?
But functional programming works, right?

;; These structs are used to represent
;; shapes in the Cartesian plain:

(struct rectangle (xmin xmax ymin ymax))
(struct circle (x y r))
(struct union (top bot))

;; Design the function shape-in?, which
;; determines whether some Cartesian
;; point is within some given shape.
A Shape is one of:
- (rectangle Int Int Int Int)
- (circle Int Int Int)
- (union Shape Shape)

Shape Int Int -> Boolean
is the point (x,y) within sh?

(define (shape-in? sh x y)
  (cond
    [(rectangle? sh) (rectangle-in? sh x y)]
    [(circle? sh) (circle-in? x y sh)]
    [(union? sh)
      (or (shape-in? (union-top sh) x y)
          (shape-in? (union-bot sh) x y))]]))
;; A Shape is one of:
;; -- (rectangle Int Int Int Int)
;; -- (circle Int Int Int)
;; -- (union Shape Shape)

;; Shape Int Int -> Boolean
;; is the point (x,y) within sh?
(define (shape-in? sh x y)
  (cond
   [(rectangle? sh) (rectangle-in? sh x y)]
   [(circle? sh) (circle-in? x y sh)]
   [(union? sh)
    (or (shape-in? (union-top sh) x y)
        (shape-in? (union-bot sh) x y))])))

Why is this good?
How did we get here?
;; A Shape is one of:
;; -- (rectangle Int Int Int Int)
;; -- (circle Int Int Int)
;; -- (union Shape Shape)

;; Shape Int Int -> Boolean
;; is the point (x,y) within sh?
(define (shape-in? sh x y)
  (match sh
    [(rectangle a b c d) (rectangle-in? sh x y)]
    [(circle xc yc r) (circle-in? x y sh)]
    [(union t b) (or (shape-in? t x y) (shape-in? b x y))])))
A Shape is one of:

-- (rectangle Int Int Int Int)
-- (circle Int Int Int)
-- (union Shape Shape)

Shape Int Int -> Boolean
is the point (x,y) within sh?
(define (shape-in? sh x y)
  (match sh
    [(rectangle a b c d) (rectangle-in? sh x y)]
    [(circle xc yc r) (circle-in? x y sh)]
    [(union t b)
      (or (shape-in? t x y)
           (shape-in? b x y))])))

Some might consider this a better solution.
A Shape is one of:

-- (rectangle Int Int Int Int)
-- (circle Int Int Int)
-- (union Shape Shape)

Shape Int Int -> Boolean  is the point (x,y) within sh?

(define (shape-in? sh x y)
  (match sh
    [(rectangle a b c d) (rectangle-in? sh x y)]
    [(circle xc yc r) (circle-in? x y sh)]
    [(union t b)
      (let ([in-t? (shape-in? t x y)]
            [in-b? (shape-in? b x y)])
        (or in-t? in-b?))])))
;; A Shape is one of:
;; -- (rectangle Int Int Int Int)
;; -- (circle Int Int Int)
;; -- (union Shape Shape)

;; Shape Int Int -> Boolean
;; is the point (x,y) within sh?
(define (shape-in? sh x y)
  (match sh
    [(rectangle a b c d) (rectangle-in? sh x y)]
    [(circle xc yc r) (circle-in? x y sh)]
    [(union t b)
      (let ([in-t? (shape-in? t x y)]
            [or in-t? (set! in-t? (shape-in? b x y))
                       in-t?)]))])

Why is this BAD? How can we fail this solution?
(defun shape-in? (sh x y)
  (match sh
    [(rectangle a b c d) (rectangle-in? sh x y)]
    [(circle xc yc r) (circle-in? x y sh)]
    [(union t b)
      (let ([in-t? (shape-in? t x y)])
        (call/cc
          (lambda (exit)
            (or in-t?
                (exit true)
                (set! in-t? (shape-in? b x y))
                in-t?))))]))
;; [Listof Posn] -> Image
;; add red circles to background-scene for all positions in lop
(define (draw* lop) ...)
;; [Listof Posn] -> Image
;; add red circles to background-scene for all positions in lop
(define (draw* lop) ...)

(define (draw* lop)
  (cond
    [(empty? lop) background-scene]
    [else (draw-one (draw* (rest lop)) (first lop))])))
;; [Listof Posn] -> Image
;; add red circles to background-scene for all positions in lop
(define (draw* lop) ...)

(define (draw* lop)
  (cond
   [(empty? lop) background-scene]
   [else (draw-one (draw* (rest lop)) (first lop))]))

(define (draw* lop)
  (foldl draw-one background-scene lop))

Why is this good?
How did we get here??
;; [Listof Number] -> [Listof Number]
;; Given a list of relative distances (lord)
;; produce a list of absolute distances
;; [Listof Number] -> [Listof Number]
;; Given a list of relative distances (lord)
;; produce a list of absolute distances

(define (absolute lord)
  (cond
    [(empty? lord) '()]  
    [else (cons (first lord)
                 (map (curry + (first lord))
                      (absolute (rest lord))))])
)

Tuesday, September 28, 2010
;; [Listof Number] -> [Listof Number]
;; Given a list of relative distances (lord)
;; produce a list of absolute distances

(define (absolute lord)
  (cond
    [(empty? lord) '()]
    [else (cons (first lord)
                  (map (curry + (first lord))
                       (absolute (rest lord))))]))

(define (absolute.v2 lord0)
  ;; accumulator: so-far is distance
  ;; between lord0 and lord
  (define (absolute lord so-far)
    (cond
      [(empty? lord) '()]
      [else (let* ([f (first lord)]
                   [d (+ f so-far)])
             (cons d (absolute (rest lord) d)))]))
  ;; -- IN --
  (absolute lord 0))

Why is this better?  
How did we get here?

Tuesday, September 28, 2010
How do we get from the problem to the solution?

Can we get our students to repeat these thought processes on their own eventually?
Process

- represent information as data
- formulate signature & purpose
- illustrate with examples
- organize a program template
- code (define the function)
- turn examples into unit tests
<table>
<thead>
<tr>
<th>Process</th>
<th>Principles</th>
</tr>
</thead>
<tbody>
<tr>
<td>* represent information as data</td>
<td>* recur structurally</td>
</tr>
<tr>
<td>* formulate signature &amp; purpose</td>
<td>* abstract from examples</td>
</tr>
<tr>
<td>* illustrate with examples</td>
<td>* abstractions from types</td>
</tr>
<tr>
<td>* organize a program template</td>
<td>* recur but generate</td>
</tr>
<tr>
<td>* code (define the function)</td>
<td>* ... with accumulators</td>
</tr>
<tr>
<td>* turn examples into unit tests</td>
<td>* ... with mutable state</td>
</tr>
<tr>
<td>Process</td>
<td>Principles</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>* represent information as data</td>
<td>* recur structurally</td>
</tr>
<tr>
<td>* formulate signature &amp; purpose</td>
<td>* abstract from examples</td>
</tr>
<tr>
<td>* illustrate with examples</td>
<td>* abstractions from types</td>
</tr>
<tr>
<td>* organize a program template</td>
<td>* recur but generate</td>
</tr>
<tr>
<td>* code (define the function)</td>
<td>* ... with accumulators</td>
</tr>
<tr>
<td>* turn examples into unit tests</td>
<td>* ... with mutable state</td>
</tr>
</tbody>
</table>

Tuesday, September 28, 2010
### Process
- represent information as data
- formulate signature & purpose
- illustrate with examples
- organize a program template
- code (define the function)
- turn examples into unit tests

### Principles
- recur structurally
- abstract from examples
- abstractions from types
- recur but generate
- ... with accumulators
- ... with mutable state
**Process**

- information as data
- signature & purpose
- illustrate with examples
- organize a template
- define the function
- examples as tests

**Structural Template**

- Does the data definition distinguish distinct subsets?
- Conditions over parameters that distinguish those?
- Do any of the lines involve compound data?
- Are there self-references in the data definitions?
Process

* information as data
* signature & purpose
* illustrate with examples
* organize a template
* define the function
* examples as tests

Structural Coding

* Tackle the base cases first (with example).
* Otherwise remind yourself what the template expressions compute.
* Combine these values. Define auxiliaries if needed.
Process

- information as data
- signature & purpose
- ... examples
- organize template
- define the function
- examples as tests

;;; EXERCISE:

;;; Use structs to represent shapes in the Cartesian plain:
;;; * rectangles, * circles,
;;; * combinations of two shapes.

;;; Design the function shape-in?,
;;; which determines whether some coordinates are within a given shape.
Process

- information as data
- signature & purpose
- ... examples
- organize template
- define the function
- examples as tests

(struct rectangle (xn xx yn yx))
(struct circle (x y r))
(struct union (top bot))

;; A Shape is one of:
;; -- (rectangle Int Int Int Int)
;; -- (circle Int Int Int)
;; -- (union Shape Shape)

;; Shape Int Int -> Boolean
;; is the point (x,y) within sh?
(define (in? sh x y)
  (cond
   [(rectangle? sh)
    (rectangle-in? sh x y)]
   [(circle? sh)
    (circle-in? x y sh)]
   [(union? sh)
    (or (in? (union-top sh) x y)
        (in? (union-bot sh) x y))])))
Process

* information as data
* signature & purpose
* ... examples
* organize template
* define the function
* examples as tests

(struct rectangle (xn xx yn yx))
(struct circle (x y r))
(struct union (top bot))

;; A Shape is one of:
;; -- (rectangle Int Int Int Int)
;; -- (circle Int Int Int)
;; -- (union Shape Shape)

;; Shape Int Int -> Boolean
;; is the point (x,y) within sh?
(define (in? sh x y)
  (cond
   [(rectangle? sh)
    (rectangle-in? sh x y)]
   [(circle? sh)
    (circle-in? x y sh)]
   [(union? sh)
    (or (in? (union-top sh) x y)
         (in? (union-bot sh) x y))])))
A Shape is one of:
-- (rectangle Int Int Int Int Int)
-- (circle Int Int Int)
-- (union Shape Shape)

Shape Int Int -> Boolean
is the point (x,y) within sh?
(3,4) is in (circle 0 0 5)
(3,4) not in (rectangle 0 1 0 1)
(3,4) in (union (circle 0 0 5) ...
Process

- information as data
- signature & purpose
- illustrate with examples
- organize a template
- define the function
- examples as tests

Template

- Does the data definition distinguish distinct subsets?
- Conditions over parameters that distinguish those?
- Do any of the lines involve compound data?
- Are there self-references in the data definitions?
Template

* ... data ... distinct subsets

* ... conditions ... parameters?

* .. compound data?

* ... self-references ... in data definition?

--; A Shape is one of:
--; -- (rectangle Int Int Int Int Int)
--; -- (circle Int Int Int)
--; -- (union Shape Shape)

--; Shape Int Int -> Boolean
--; is the point (x,y) within sh?
--; (3,4) is in (circle 0 0 5)
--; (3,4) not in (rectangle 0 1 0 1)
--; (3,4) in (union (circle 0 0 5) ...
(define (in? sh x y)
  (cond
    [(rectangle? sh)
      (rectangle-in? sh x y)]
    [(circle? sh)
      (circle-in? x y sh)]
    [(union? sh)
      (or (in? (union-top sh) x y)
          (in? (union-bot sh) x y))])

Tuesday, September 28, 2010
* ... data ... distinct subsets

* ... conditions ... parameters?

* .. compound data?

* ... self-references ... in data definition?

---

```
;; A Shape is one of:
;; -- (rectangle Int Int Int Int Int Int)
;; -- (circle Int Int Int Int)
;; -- (union Shape Shape)

;; Shape Int Int -> Boolean
;; is the point (x,y) within sh?
;; (3,4) is in (circle 0 0 5)
;; (3,4) not in (rectangle 0 1 0 1)
;; (3,4) in (union (circle 0 0 5) ... (define (in? sh x y)
  (cond
    [(rectangle? sh)
      (rectangle-in? sh x y)]
    [(circle? sh)
      (circle-in? x y sh)]
    [(union? sh)
      (or (in? (union-top sh) x y)
          (in? (union-bot sh) x y))])
```

Tuesday, September 28, 2010
Template

* ... data ... distinct subsets
* ... conditions ... parameters?
* .. compound data?
* ... self-references ... in data definition?

;; A Shape is one of:
;; -- (rectangle Int Int Int Int Int)
;; -- (circle Int Int Int)
;; -- (union Shape Shape)

;; Shape Int Int -> Boolean
;; is the point (x,y) within sh?
;; (3,4) is in (circle 0 0 5)
;; (3,4) not in (rectangle 0 1 0 1)
;; (3,4) in (union (circle 0 0 5) ...
(define (in? sh x y)
  (cond
    [(rectangle? sh) (rectangle-x sh)
     ... (rectangle-y sh) ...]
    [(circle? sh) (circle-x sh) ...
     (circle-y sh) ... (circle-r sh)]
    [(union? sh)
       (or (in? (union-top sh) x y)
           (in? (union-bot sh) x y))]]))
... data ... distinct subsets

... conditions ... parameters?

.. compound data?

... self-references ... in data definition?

---

```scheme
;; A Shape is one of:
;; -- (rectangle Int Int Int Int Int)
;; -- (circle Int Int Int)
;; -- (union Shape Shape)

;; Shape Int Int -> Boolean
;; is the point (x,y) within sh?
;; (3,4) is in (circle 0 0 5)
;; (3,4) not in (rectangle 0 1 0 1)
;; (3,4) in (union (circle 0 0 5) ...
(define (in? sh x y)
  (cond
    [(rectangle? sh) (rectangle-x sh)
      ... (rectangle-y sh) ...]
    [(circle? sh) (circle-x sh) ...
      (circle-y sh) ... (circle-r sh)]
    [(union? sh)
      (or (in? (union-top sh) x y)
        (in? (union-bot sh) x y))]]))
```
Template

* information as data
* signature & purpose
* ... examples
* organize template
* define the function
* examples as tests

;;; A Shape is one of:
;;; -- (rectangle Int Int Int Int Int)
;;; -- (circle Int Int Int)
;;; -- (union Shape Shape)

;;; Shape Int Int -> Boolean
;;; is the point (x,y) within sh?
;;; (3,4) is in (circle 0 0 5)
;;; (3,4) not in (rectangle 0 1 0 1)
;;; (3,4) in (union (circle 0 0 5) ... (define (in? sh x y)
  (cond
    [(rectangle? sh)
      (rectangle-in? sh x y)]
    [(circle? sh)
      (circle-in? x y sh)]
    [(union? sh)
      (or (in? (union-top sh) x y)
          (in? (union-bot sh) x y))])}}
Template

* information as data
* signature & purpose
* ... examples
* organize template
* define the function
* examples as tests

```scheme
;; A Shape is one of:
;; ... 

;; Shape Int Int -> Boolean
;; is the point (x,y) within sh?
(define (in? sh x y)
  (cond
    [(rectangle? sh)
      (rectangle-in? sh x y)]
    [(circle? sh)
      (circle-in? x y sh)]
    [(union? sh)
      (or (in? (union-top sh) x y)
      (in? (union-bot sh) x y))]])

(check-expect
  (in? (circle 0 0 5) 3 4) true)
(check-expect
  (in? (union ... ... ) 3 4) true)
```
Process

- information as data
- signature & purpose
- illustrate with examples
- organize a template
- define the function
- examples as tests

How it all helps ...

where are you stuck?

Tuesday, September 28, 2010
Process

- information as data
- signature & purpose
- illustrate with examples
- organize a template
- define the function
- examples as tests

How it all helps ...

where are you stuck?
Process

* information as data
* signature & purpose
* illustrate with examples
* organize a template
* define the function
* examples as tests

How it all helps ...

The design process and principles are also pedagogic diagnostic tools that help students help themselves.
Principles

- recur structurally
- abstract from examples
- abstractions from types
- recur with generate
- ... with accumulators
- ... with mutable state

how does it work here?
Principles

- recur structurally
- abstract from examples
- abstractions from types
- recur with generate
- ... with accumulators
- ... with mutable state

how does it work here?
How the *Process* applies to abstract: the template

```scheme
;; [Listof Number] -> [Listof Boolean]
(define (good lon)
  (cond
    [(empty? lon) '()]
    [else (cons (negative? (first lon)) (good (rest lon)))]))

;; [Listof PersonnelRecord] -> [Listof Salary]
(define (salary* lon)
  (cond
    [(empty? lon) '()]
    [else (cons (pr-dat (first lon)) (salary* (rest lon)))]))
```
;; [Listof Number] -> [Listof Boolean]
(define (good lon)
  (cond
   [(empty? lon) '()]  
   [else (cons (negative? (first lon)) (good (rest lon))))]))

;; [Listof PersonnelRecord] -> [Listof Salary]
(define (salary* lon)
  (cond
   [(empty? lon) '()]  
   [else (cons (pr-dat (first lon)) (salary* (rest lon))))]))
How the Process applies to abstract: the coding & testing

;; [Listof X] [X -> Y] -> [Listof Y]
(define (map lon f)
  (cond
    [(empty? lon) '()]
    [else (cons (f (first lon)) (map (rest lon) f))]))

;; tests
(define (good lon) (map lon negative?))
(define (salary* lon) (map lon pr-salary))
Principles

* recur structurally
* abstract from examples
* abstractions from types
* recur with generate
* ... with accumulators
* ... with mutable state

the design process works everywhere
but there is a problem:
if you accept *design principle*, the old ideas on freshman course don’t work
Principles

* recur structurally
* abstract from examples
* abstractions from types
* generative recursion
* ... with accumulators
* ... with mutable state

It replaces Turing power (and Felleisen power) with design.
Principles

- recur structurally
- abstract from examples
- abstractions from types
- generative recursion
- ... with accumulators
- ... with mutable state

It replaces Turing power (and Felleisen power) with design.

Turing and Felleisen power is *large enough.*
Principles

- recur structurally
- abstract from examples
- abstractions from types
- generative recursion
- ... with accumulators
- ... with mutable state

It replaces Turing power (and Felleisen power) with design.

Turing and Felleisen power is large enough.

... but students can’t design accumulator-style yet
Principles

- recur structurally
- abstract from examples
- abstractions from types
- generative recursion
- ... with accumulators
- ... with mutable state

even with higher-order functions, it is impossible to design QUICKSORT
Principles

- recur structurally
- abstract from examples
- abstractions from types
- generative recursion
  - ... with accumulators
  - ... with mutable state

even with higher-order functions, it is impossible to design a terminating GRAPH TRAVERSAL
Design for the Working SE

systematic procedure
= how to design

K-12

Design for the Working SE

systematic procedure
= how to design

K-12

Func. Prog.


how about upstream?

Tuesday, September 28, 2010
Kids want to have fun, but they are willing to learn more than teachers grant.
Kids want to have fun, but they are willing to learn more than teachers grant.

;; ClockTicks -> Image
(define (flying-soccer-ball t)
    (place-image X0 (y t) BACKGROUND))

;; run program run:
(animate flying-soccer-ball)
Kids want to have fun, but they are willing to learn more than teachers grant.

```
;; ClockTicks -> Image
(define (flying-soccer-ball t)
  (place-image X0 (y t) BACKGROUND))

;; run program run:
(animate flying-soccer-ball)
```

```
(big-bang GameState0
  (to-draw ;; GameState -> Image
    render-state-as-image)
  (on-tick ;; GameState -> GameState
    handle-clock-tick)
  (on-key ;; GameState KeyEvent -> GameState
    handle-key-event)
  ...)
```
Kids want to have fun, but they are willing to learn more than teachers grant.

- images as a datatype
- functional animations
- functional interactive games

Felleisen, Findler, Flatt, Krishnamurthi
A Functional I/O System
ICFP 2009, Edinburgh
Kids want to have fun, but they are willing to learn more than teachers grant.

- algebra, n-ary functions
- geometry & trigonometry
- even pre-calculus
Design for the Working SE

systematic procedure = how to design

helping kids, helps freshman

Design for the Working SE

systematic procedure = how to design

helping kids, helps freshman


K-12
How to Design ... Worlds
1. Identify those properties that change and those that remain the same.
1. Identify those properties that change and those that remain the same.

2. Formulate data definitions (GameState) for properties that change and constant definitions for those that don’t.

This creates a function wish list:

```plaintext
;; tickh: GameState -> GameState
;; keyh: GameState KeyEvent -> GameState
;; mouseh: GameState ... -> GameState
...```
1. Identify those properties that change and those that remain the same.

2. Formulate data definitions (GameState) for properties that change and constant definitions for those that don’t.

This creates a function wish list:

```haskell
;; tickh: GameState -> GameState
;; keyh: GameState KeyEvent -> GameState
;; mouseh: GameState ... -> GameState
...```

3. Associate game actions (movement of ball) to world actions (clock tick, key handler, mouse event, etc).

Worlds provide a place to start, and a wish list to work off.
• Program design driven by wish lists.

• Functions organized around data (OO).

• Mostly structural designs.
Design for the Working SE

systematic procedure = how to design, but don’t forget some fun
Design for the Working SE

systematic procedure = how to design, but don't forget some fun

K-12


Func. Prog.

how about downstream?

Tuesday, September 28, 2010
what if a student must use JAVA or PYTHON on the first co-op?
what if a student must use JAVA or PYTHON on the first co-op?

what can we teach students that we couldn’t have taught them before?
HtDP

structure abstraction
generative accumulator

Downstream:
how to connect to industrial practice (OOP)

HtDC

structure ~ class organization
structure & state
abstraction ~ inheritance & overriding
abstraction ~ traversals
abstraction ~ generics

design ~ OO patterns
iteration vs OO methods
APIs & iteration
Downstream: how to connect to industrial practice (OOP)

Problem isolated, but not solved
Downstream @ Northeastern introduce freshmen to theorem proving about interesting programs.

HtDP
- structure
- animation
- abstraction
- games
- generative
- client & server
- accumulator

Logic and Program Design
- unit tests ~ conjectures: QuickCheck
- conjectures ~ explicit proofs & theorems
- games ~ automatic theorem proving
- structure ~ automatic theorem proving

theorems:
- games ~ virtual machines
- ACL2 isn’t Racket: lack of safety
- ACL2 isn’t Racket: lack of modules
- ACL2 isn’t Racket: lack of HO functions
Downstream @ Northeastern introduce freshmen to theorem proving about interesting programs

- unit tests ~ conjectures: QuickCheck
- conjectures ~ explicit proofs & theorems
- games ~ automatic theorem proving
- structure ~ automatic theorem proving
- theorems: games ~ virtual machines
- ACL2 isn’t Racket: lack of safety
- ACL2 isn’t Racket: lack of modules
- ACL2 isn’t Racket: lack of HO functions
Summary

How to Design Classes

Logic and Program Design

How to Design Programs

Freshman Year
Tuesday, September 28, 2010
Failures & Challenges
Teaching languages matter.
The IDE matters.
Teaching languages matter.
The IDE matters.

We failed to bring across how much they matter:
Crestani & Sperber, ICFP '10
Teaching languages matter.
The IDE matters.

We failed to bring across how much they matter:
*Crestani & Sperber, ICFP ’10*

We failed to evaluate them scientifically and systematically:
*Marceau, Fisler, & Krishnamurthi, SFP ’10*
Changing the curriculum is difficult.
A new effort has one shot only.
Changing the curriculum is difficult.
A new effort has one shot only.

Choose your name wisely.
To this day people think our work is about *Scheme*. 
Changing the curriculum is difficult.  
A new effort has one shot only.

Choose your name wisely.  
To this day people think our work is about \textit{Scheme}.

Put your best face forward.  
Functional animations came way tooooooo late.
Changing the curriculum is difficult. A new effort has one shot only.

Choose your name wisely. To this day people think our work is about *Scheme*.

Put your best face forward. Functional animations came way tooooooo late.

Get consumers involved. Appreciate *all* contributions from teachers.
All computer science curricula start from a course on the currently fashionable programming language.
All computer science curricula start from a course on the currently fashionable programming language.

Every first course must be about a programming language. Yours must be about Scheme. What’s this DESIGN stuff?
All computer science curricula start from a course on the currently fashionable programming language.

Every first course must be about a programming language. Yours must be about Scheme. What’s this DESIGN stuff?

Our curriculum requires whole-sale adoption. It is nearly impossible to integrate bits and pieces.
Where are the for loops?

It is not about imperative vs functional programming.

It is about inductive vs indexed programming.
Where are the for loops?

It is not about imperative vs functional programming.

It is about inductive vs indexed programming.

Challenge: we need a design recipe for dealing with indexed data structures well.
Our community is you. Getting involved is good for the kids and good for you.

Get our community involved.
The Virtuous Cycle of Teaching & Research

Teaching → Research

Tuesday, September 28, 2010
The Virtuous Cycle of Teaching & Research

It’s not about Racket, it’s about FP ideas

It’s about changing Comp Sci

And we can help others, too
The Virtuous Cycle of Teaching & Research

It's about changing Comp Sci
And we can help others, too

It's not about Racket, it's about FP ideas

the language is the OS (ICFP ’00)
higher-order contracts (ICFP ’02)
modularized macros (ICFP ’02)
slideshow (ICFP ’05)

JVM continuations (ICFP ’07)
scribbling manuals (ICFP ’09)
functional (gui) i/o (ICFP ’09)
logical occurrence typing (ICFP ’10)
fortified macros (ICFP ’10)
Conclusions
• Design is a topic.

• Teaching is a means to explore design. And it helps people.

• Research follows naturally.
• Design is a topic.

• Teaching is a means to explore design. And it helps people.

• Research follows naturally.

Design is a big map with lots of white space.
• Design is a topic.
• Teaching is a means to explore design. And it helps people.
• Research follows naturally.

Design is a big map with lots of white space.
The End

(in order of appearance)
The End
(in order of appearance)

Matthew Flatt
Shriram Krishnamurthi
Robby Findler
Kathi Fisler
John Clements
Paul Graunke
Kathy Gray
Scott Owen
Jacob Matthews
Adam Wick
Paul Steckler
Philippe Meunier

Eli Barzilay
Jay McCarthy
Mike Sperber
Sam Tobin-Hochstadt
Carl Eastlund
Ryan Culpepper
Stevie Strickland
Casey Klein
Christos Dimoulas
Kevin Tew
Jon Rafkind
Danny Yoo
Guillaume Marceau
Vincent St. Amour
James Swaine
Stephen Chang

Tuesday, September 28, 2010