If you view this talk in PowerPoint, turn on comments (View | Comments in PowerPoint) to read remarks made during the talk but not included on the slide.

Once you do this, you ought to see a comment attached to this slide (outside presentation mode).
In your favorite C++ environment:

\[ \text{wage\_per\_hour} \times \text{number\_of\_hours} = \text{total\_wage} \]
Current Programming Practice

In your favorite C++ environment:

\[ \text{wage\_per\_hour} \times \text{number\_of\_hours} = \text{total\_wage} \]

pointer manipulation
In your favorite Java environment:

class Main {
    public static void main(String args[]) {
        if (args.length != 1) {
            throw new InvalidInput("This program needs one argument.");
        }
        System.out.println("You entered " + args[0]);
    }
    class InvalidInput extends RuntimeException {
        public InvalidInput(String mess) { super(mess); }
    }
}
In your favorite Java environment:

class Main {
    public static void main(String args[]) {
        if (args.length != 1) {
            throw new InvalidInput("This program needs one argument.");
        }
        System.out.println("You entered " + args[0]);
    }
    class InvalidInput extends RuntimeException {
        public InvalidInput(String mess) { super(mess); }
    }
}
In Pascal/Java/C++:

\[
i = 0;
\text{do } \{
\quad \text{read (j);
\quad \text{if (j > 0)}
\quad \quad i = i + j;
\}\}
\text{while (j > 0)}
\]
In Pascal/Java/C++:

```c
i = 0;
do {
    read (j);
    if (j > 0)
        i = i + j;
} while (j > 0)
```

The sum of a sequence of positive numbers is
- positive
- zero
- negative
;;; length2 : list -> number
(define (length2 alox)
  (cond
    [empty?(alox) 0]
    [else (+ 1 (length2 (rest alox)))]))

Welcome to MzScheme version 102/16, Copyright (c) 1995-2000 PLT (Matthew Flatt)
Identifiers and symbols are initially case-sensitive.

> > length2
#<procedure:length2>
> (length2 '())
procedure application: expected procedure, given: () (no arguments)
> (length2 '(aaa bbb ccc))
procedure application: expected procedure, given: (aaa bbb ccc) (no arguments)
> []
an error message with no error locus
an error message with no error locus
(define (length1 alox)
  (cond
    [(empty? alox) 0]
    [else 1 + (length1 (rest alox))])))
;;; length1 : list -> number
(define (length1 alox)
  (cond
   [(empty? alox) 0]
   [else 1 + (length1 (rest alox))])))

Welcome to MzScheme version 102/16, Copyright (c) 1995-2000 PLT (Matthew Flatt)
Identifiers and symbols are initially case-sensitive.
> length1
#<procedure:length1>
> (length1 '())
0
> (length1 '(aaa bbb ccc))
0
> (length1 '(aaa bbb ccc ddd eee fff ggg hhh ill))
0

length1 returns 0, no matter what input
length1 : list -> number
(define (length1 alox)
  (cond
   [(empty? alox) 0]
   [else 1 + (length1 (rest alox))])))

length1 returns 0, no matter what input
How to Produce the Best OO Programmers

Shriram Krishnamurthi
Brown University
and
The TeachScheme! Project
Current Practice in Introductory Courses

• Teach the syntax of a currently fashionable programming language

• Use Emacs or commercial PE

• Show examplars of code and ask students to mimic

• Discuss algorithmic ideas (O(-))
Current Practice: Design vs Tinkering

• Syntax: too complex; must tinker

• Design: exposition of syntactic constructs takes the place of design guidelines

• Teaching standard algorithms doesn’t replace a discipline of design
Lessons: The Trinity

• Simple programming language

• Programming environment for students

• A discipline of design
  – algorithmic sophistication follows from design principles
How to Design Programs
(methodology)

Scheme (language)
DrScheme (environment)
TeachScheme! is not MIT's Scheme!
TeachScheme! is not MIT’s Scheme!

- Cleans up the MIT’s Scheme language
TeachScheme! is not MIT’s Scheme!

- Cleans up the MIT’s Scheme language
- Not MIT’s programming environment
TeachScheme! is not MIT’s Scheme!

• Cleans up the MIT’s Scheme language
• Not MIT’s programming environment
• Most importantly: not SICP pedagogy
TeachScheme! is not MIT’s Scheme!

- Cleans up the MIT’s Scheme language
- Not MIT’s programming environment
- Most importantly: not SICP pedagogy
  - fails the normal student
TeachScheme! is not MIT’s Scheme!

• Cleans up the MIT’s Scheme language

• Not MIT’s programming environment

• Most importantly: not SICP pedagogy
  – fails the normal student
  – does not discuss program design
TeachScheme! is not MIT’s Scheme!

- Cleans up the MIT’s Scheme language
- Not MIT’s programming environment
- Most importantly: not SICP pedagogy
  - fails the normal student
  - does not discuss program design
  - has an outdated idea of OO programming
TeachScheme! is not MIT’s Scheme!

- Cleans up the MIT’s Scheme language
- Not MIT’s programming environment
- Most importantly: not SICP pedagogy
  - fails the normal student
  - does not discuss program design
  - has an outdated idea of OO programming
  - ignores applications and other attractions
Part I:
Programming Language
Programming Language: Scheme
• Scheme’s notation is simple:
  – either atomic or (<op> <arg> …)
  
• 3  (+ 1 2)  (+ (* 3 4) 5)  (* (/ 5 9) (- t 32))
• Scheme’s notation is simple:
  – either atomic or \((\text{<op>} \ \text{<arg>} \ ...\)\)
    • 3 \((+\ 1\ 2)\) \((+\ (*\ 3\ 4)\ 5)\) \((*/\ 5\ 9)\ (-\ t\ 32))\)

• Scheme’s semantics is easy:
  – it’s just the rules of algebra
    • no fussing with calling conventions, compilation models, stack frames, activation records, etc.
    • exploits what students already know
**Programming Language: Scheme**

- **Scheme’s notation is simple:**
  - either atomic or \((<\text{op}> \ <\text{arg}> \ldots)\)
    - \(3\) \((+ 1 2)\) \((+ (* 3 4) 5)\) \((* (/ 5 9) (- t 32))\)

- **Scheme’s semantics is easy:**
  - it’s just the rules of algebra
    - no fussing with calling conventions, compilation models, stack frames, activation records, etc.
    - exploits what students already know

- **With Scheme, we can focus on ideas**
Learning the Language
Learning the Language

• Students write full programs from the first minute
Learning the Language

• **Students write full programs from the first minute**

• **Only five language constructs introduced in the entire semester**
Learning the Language

- Students write full programs from the first minute
- Only five language constructs introduced in the entire semester
- Takes < 1 week to adapt to prefix
  - no need to memorize precedence tables!
And Yet ...

• Simple notational mistakes produce
  – error messages beyond the students’ knowledge
  – strange results -- without warning

• ... and even in Scheme (let alone Java/C++/etc.) there are just too many features
Programming Languages: Not One, Many
Programming Languages: Not One, Many

• **Language 1: first-order, functional**
  – function definition and application
  – conditional expression
  – structure definition
Programming Languages: Not One, Many

• **Language 1: first-order, functional**
  – function definition and application
  – conditional expression
  – structure definition

• **Language 2: local function definitions**
Programming Languages: Not One, Many

• **Language 1:** first-order, functional
  - function definition and application
  - conditional expression
  - structure definition

• **Language 2:** local function definitions

• **Language 3:** functions and effects
  - higher-order functions
  - mutation and sequencing
Programming Languages
Programming Languages

• Layer language by pedagogic needs
Programming Languages

- Layer language by pedagogic needs
- Put students in a knowledge-appropriate context
Programming Languages

• Layer language by pedagogic needs

• Put students in a knowledge-appropriate context

• Focus on design relative to context
Programming Languages

- Layer language by pedagogic needs
- Put students in a knowledge-appropriate context
- Focus on design relative to context

Result of over five years of design
Part II: Programming Environment
Programming Environment Desiderata
Programming Environment Desiderata

• Enforce all language levels
Programming Environment Desiderata

- Enforce all language levels
- Safe, so errors are trapped
Programming Environment Desiderata

- Enforce all language levels
- Safe, so errors are trapped
- Highlight location of dynamic errors
Programming Environment Desiderata

- Enforce all language levels
- Safe, so errors are trapped
- Highlight location of dynamic errors
- Enable instructors to provide code 
  not at student’s level
Programming Environment Desiderata

• Enforce all language levels
• *Safe*, so errors are trapped
• **Highlight location of dynamic errors**
• Enable instructors to provide code *not* at student’s level
• **Facilitate interactive exploration**
Programming Environment Desiderata

• Enforce all language levels
• Safe, so errors are trapped
• Highlight location of *dynamic* errors
• Enable instructors to provide code *not* at student’s level
• Facilitate interactive exploration
• Cross-platform compatibility
• Enforce all language levels
• Safe, so errors are trapped
• Highlight location of *dynamic* errors
• Enable instructors to provide code *not* at student’s level
• Facilitate interactive exploration
• Cross-platform compatibility
• How about a “Break” button?
Some of DrScheme’s Features
Some of DrScheme’s Features

• Layer-oriented languages and errors
Some of DrScheme’s Features

• Layer-oriented languages and errors
• Highlighting of dynamic errors
Some of DrScheme’s Features

- Layer-oriented languages and errors
- Highlighting of dynamic errors
- Explanation of scoping rules
Some of DrScheme’s Features

- Layer-oriented languages and errors
- Highlighting of dynamic errors
- Explanation of scoping rules
- Algebraic stepper
Some of DrScheme’s Features

- Layer-oriented languages and errors
- Highlighting of dynamic errors
- Explanation of scoping rules
- Algebraic stepper
- Interesting values (even pictures)
Some of DrScheme’s Features

- Layer-oriented languages and errors
- Highlighting of dynamic errors
- Explanation of scoping rules
- Algebraic stepper
- Interesting values (even pictures)
- Teachpacks
Some of DrScheme’s Features

- Layer-oriented languages and errors
- Highlighting of dynamic errors
- Explanation of scoping rules
- Algebraic stepper
- Interesting values (even pictures)
- Teachpacks
- cross-platform GUIs, networking, etc.
Some of DrScheme’s Features

- Layer-oriented languages and errors
- Highlighting of dynamic errors
- Explanation of scoping rules
- Algebraic stepper
- Interesting values (even pictures)
- Teachpacks
- cross-platform GUIs, networking, etc.
- Oh, and that “Break” button
Part III:
Design Methodology
• Implicitly foster basic good habits

• Rational in its design
  – its steps explain the code’s structure

• Accessible to beginner
Design Recipes

to be designed
Design Recipes

How do we wire the “program” to the rest of the world?
Design Recipes

How do we wire the “program” to the rest of the world?

IMPERATIVE: Teach Model-View Separation
Given data, the central theme:

**Data Drive Designs**

From the structure of the data, we can derive the basic structure of the program ...

So let’s do!
Design Recipes: Class Definitions

- use rigorous but not formal language
- start with the familiar
  - basic sets: numbers, symbols, booleans
  - intervals on numbers
- extend as needed
  - structures
  - unions
  - self-references
  - vectors (much later)
Consider the lowly armadillo:
• it has a name
• it may be alive (but in Texas ...)

Design Recipes: Class Definitions (2)
Consider the lowly armadillo:

- it has a name
- it may be alive (but in Texas ...)

(define-struct armadillo (name alive?))

An *armadillo* is a structure:

(make-armadillo symbol boolean)
A *zoo animal* is either
- an armadillo, or
- a tiger, or
- a giraffe

Each of these classes of animals has *its own definition*
A list-of-zoo-animals is either
A list-of-zoo-animals is either
  • empty
A list-of-zoo-animals is either
• empty
• (cons animal list-of-zoo-animals)
A list-of-zoo-animals is either
• empty
• (cons animal list-of-zoo-animals)
A list-of-zoo-animals is either
• empty
• (cons animal list-of-zoo-animals)

Let’s make examples:
A \textit{list-of-zoo-animals} is either

- empty
- \texttt{(cons animal list-of-zoo-animals)}

Let’s make examples:

- empty
A list-of-zoo-animals is either
• empty
• (cons animal list-of-zoo-animals)

Let’s make examples:
• empty
• (cons (make-armadillo ’Bubba true) empty)
A list-of-zoo-animals is either
  • empty
  • (cons animal list-of-zoo-animals)

Let’s make examples:
  • empty
  • (cons (make-armadillo ’Bubba true) empty)
  • (cons (make-tiger ’Tony ’FrostedFlakes)
A \textit{list-of-zoo-animals} is either
\begin{itemize}
\item empty
\item (cons \textit{animal} \textit{list-of-zoo-animals})
\end{itemize}

Let’s make examples:
\begin{itemize}
\item empty
\item (cons (make-armadillo ’Bubba true) empty)
\item (cons (make-tiger ’Tony ’FrostedFlakes) (cons (make-armadillo … …))
A list-of-zoo-animals is either
• empty
• (cons animal list-of-zoo-animals)

Let’s make examples:
• empty
• (cons (make-armadillo ’Bubba true) empty)
• (cons (make-tiger ’Tony ’FrostedFlakes)
  (cons (make-armadillo … …) empty))
A list of zoo animals is either
  • empty
  • (cons animal a-list-of-zoo-animals)

;; fun-for-zoo : list-of-zoo-animals -> ???
(define (fun-for-zoo a-loZA) … )
A list of zoo animals is either
  • empty
  • (cons animal a-list-of-zoo-animals)

;;; fun-for-zoo : list-of-zoo-animals -&gt; ???
(define (fun-for-zoo a-loZA) … )
A list of zoo animals is either
  • empty
  • (cons animal a-list-of-zoo-animals)

;; fun-for-zoo : list-of-zoo-animals -> ???
(define (fun-for-zoo a-loZA)
  (cond
   [ [<<condition>>] <<answer>> ]
   [ [<<condition>>] <<answer>> ])))
A list of zoo animals is either
- empty
- (cons animal a-list-of-zoo-animals)

;; fun-for-zoo : list-of-zoo-animals -> ????
(define (fun-for-zoo a-loZA)
  (cond
   [ [ <<condition>>  <<answer>> ]
   [ [ <<condition>>  <<answer>> ]]))

what are the sub-classes?
A *list of zoo animals* is either

- empty
- `(cons animal a-list-of-zoo-animals)`

;; fun-for-zoo : list-of-zoo-animals -> ???
(define (fun-for-zoo a-loZA)
  (cond
   [ (empty? a-loZA) <<answer>> ]
   [ (cons? a-loZA) <<answer>> ]))
A list of zoo animals is either
- empty
- (cons animal a-list-of-zoo-animals)

;; fun-for-zoo : list-of-zoo-animals -> ???
(define (fun-for-zoo a-loZA)
  (cond
    [ (empty? a-loZA) <<answer>> ]
    [ (cons? a-loZA) <<answer>> ]))

are any of the inputs structures?
Design Recipes: Templates

A list of zoo animals is either

- empty
- (cons animal a-list-of-zoo-animals)

;; fun-for-zoo : list-of-zoo-animals -> ???
(define (fun-for-zoo a-loZA)
  (cond
    [ (empty? a-loZA) … ]
    [ (cons? a-loZA) … (first a-loZA) …
      … (rest a-loZA) … ]))
A list of zoo animals is either
• empty
• \((\text{cons animal a-list-of-zoo-animals})\)

;; fun-for-zoo : list-of-zoo-animals -> ???
(define (fun-for-zoo a-loZA)
  (cond
   [ (empty? a-loZA) … ]
   [ (cons? a-loZA) … (first a-loZA) …
     … (rest a-loZA) … ]))
A *list of zoo animals* is either

- empty
- (cons animal a-list-of-zoo-animals)

;;; fun-for-zoo : list-of-zoo-animals -> ???
(define (fun-for-zoo a-loZA)
  (cond
    [ (empty? a-loZA) … ]
    [ (cons? a-loZA) … (first a-loZA) …
      … (rest a-loZA) … ]))
Design Recipes: Templates
• A template reflects the structure of the class definitions (mostly for input, often for input)
A template reflects the structure of the class definitions (mostly for input, often for input)

This match helps designers, readers, modifiers, maintainers alike
Design Recipes: Templates

• A template reflects the structure of the class definitions (mostly for input, often for input)

• This match helps designers, readers, modifiers, maintainers alike

• Greatly simplifies function definition
;; fun-for-zoo : list-of-zoo-animals -> ???
(define (fun-for-zoo a-loZA)
  (cond
   [ (empty? a-loZA) … ]
   [ (cons? a-loZA) … (first a-loZA) …
     … (fun-for-zoo (rest a-loZA)) … ]))
;; fun-for-zoo : list-of-zoo-animals -> ???
(define (fun-for-zoo a-loZA)
   (cond
    [ (empty? a-loZA) … ]
    [ (cons? a-loZA) … (first a-loZA) …
      … (fun-for-zoo (rest a-loZA)) … ]))

;; zoo-size : list-of-zoo-animals -> number
(define (zoo-size a-loZA)
   (cond
    [ (empty? a-loZA) 0 ]
    [ (cons? a-loZA) (+ 1 (zoo-size (rest a-loZA)))]))
;; fun-for-zoo : list-of-zoo-animals -> ???
(define (fun-for-zoo a-loZA)
  (cond
   [ (empty? a-loZA) … ]
   [ (cons? a-loZA) … (first a-loZA) …
     … (fun-for-zoo (rest a-loZA)) … ]))
Defining Functions

;; fun-for-zoo : list-of-zoo-animals -> ???
(define (fun-for-zoo a-loZA)
  (cond
   [ (empty? a-loZA) … ]
   [ (cons? a-loZA) … (first a-loZA) …
     … (fun-for-zoo (rest a-loZA)) … ]))

;; has-armadillo? : list-of-zoo-animals -> boolean
(define (has-armadillo? a-loZA)
  (cond
   [ (empty? a-loZA) false ]
   [ (cons? a-loZA) (or (armadillo? (first a-loZA))
                          (has-armadillo? (rest a-loZA))) ]))
Design Recipes: Defining Functions
• Templates remind students of all the information that is available
  – which cases
  – which field values, argument values
  – what natural recursions can compute
Design Recipes: Defining Functions

• Templates remind students of all the information that is available
  – which cases
  – which field values, argument values
  – what natural recursions can compute

• The act of a function definition is
  – to choose which computations to use
  – to combine the resulting values
The Design Recipe
The Design Recipe

- data analysis and class definition
The Design Recipe

- data analysis and class definition
- contract, purpose statement, header
The Design Recipe

• data analysis and class definition

• contract, purpose statement, header

• in-out (effect) examples
The Design Recipe

- data analysis and class definition
- contract, purpose statement, header
- in-out (effect) examples
- function template
The Design Recipe

- data analysis and class definition
- contract, purpose statement, header
- in-out (effect) examples
- function template
- function definition
The Design Recipe

- data analysis and class definition
- contract, purpose statement, header
- in-out (effect) examples
- function template
- function definition
- testing, test suite development
The Design Recipe

- data analysis and class definition
- contract, purpose statement, header
- in-out (effect) examples
- function template
- function definition
- testing, test suite development
The Design Recipe

- data analysis and class definition
- contract, purpose statement, header
- in-out (effect) examples
- function template
- function definition
- testing, test suite development
Template Construction

- basic data, intervals of numbers
- structures
- unions
- self-reference, mutual references
- circularity
Which sorting method to teach first?

- Selection sort
- Insertion sort
- Quicksort
- Heap sort
- Mergesort
- Bubble sort
- ...

Intermezzo
Generative recursion: the recursive sub-problem is determined dynamically rather than statically.
Special Topic: Generative Recursion

Generative recursion: the recursive sub-problem is determined dynamically rather than statically

• What is the base case?
Generative recursion: the recursive sub-problem is determined dynamically rather than statically

• What is the base case?
• What ensures termination?
Generative recursion: the recursive sub-problem is determined dynamically rather than statically

• What is the base case?
• What ensures termination?
• Who provides the insight?
Generative recursion: the recursive sub-problem is determined dynamically rather than statically

- What is the base case?
- What ensures termination?
- Who provides the insight?

Special case: not reusable!
Special Topic: Abstraction

Factor out commonalities in
• contracts
  – corresponds to parametric polymorphism
• function bodies
  – leads to inheritance and overriding
• Get students used to discipline from the very first day

• Use scripted question-and-answer game until they realize they can do it on their own

• Works especially well for structural solutions
Part IV: From Scheme to Java

or,

“But What Does All This Have to do With OOP?”
• focus: objects and method invocation

• basic operations:
  – creation
  – select field
  – mutate field

• select method via "polymorphism"
• **focus: objects and method invocation**

• **basic operations:**
  – creation
  – select field
  – mutate field

• **select method via “polymorphism”**

• **structures and functions**
Scheme to Java: OO Computing

- **focus: objects and method invocation**
  - basic operations:
    - creation
    - select field
    - mutate field
  - select method via "polymorphism"

- **structures and functions**
  - basic operations:
    - creation
    - select field
    - mutate field
    - recognize kind
Scheme to Java: OO Computing

- **focus:** objects and method invocation
- **basic operations:**
  - creation
  - select field
  - mutate field
- **select method via “polymorphism”**

- **structures and functions**
- **basic operations:**
  - creation
  - select field
  - mutate field
  - recognize kind
- **f(o) becomes o.f()**
Scheme to Java: OO Programming

- develop class and interface hierarchy
- allocate code of function to proper subclass
• develop class and interface hierarchy

• allocate code of function to proper subclass

• develop class definitions
Scheme to Java: OO Programming

- develop class and interface hierarchy
- allocate code of function to proper subclass
- develop class definitions
- allocate code of function to proper conditional clause
A list of zoo animals is either

- empty
- (cons animal a-list-of-zoo-animals)
A list of zoo animals is either
• empty
• (cons animal a-list-of-zoo-animals)
Scheme to Java: Code Allocation

;; fun-for-zoo : list-of-zoo-animals -> ????
(define (fun-for-zoo a-loZA)
  (cond
   [ (empty? a-loZA) ]
   [ (cons? a-loZA) … (first a-loZA) …
     … (rest a-loZA) … ]))
Scheme to Java: Code Allocation

;; fun-for-zoo : list-of-zoo-animals -> ???
(define (fun-for-zoo a-loZA)
  (cond
    [(empty? a-loZA)]
    [(cons? a-loZA) ... (first a-loZA) ... (rest a-loZA) ...])

List of zoo animals

Empty:

Cons:
  animal
  list of zoo animals
Design recipes work identically to produce well-designed OO programs

The differences are notational

The differences are instructive

The resulting programs use standard design patterns
Why not just Java first?
Why not just Java first?

- Complex notation, complex mistakes
Why not just Java first?

- Complex notation, complex mistakes
- No PE supports stratified Java
Why not just Java first?

• Complex notation, complex mistakes

• No PE supports stratified Java

• Design recipes drowned in syntax
Why not just Java first?

- Complex notation, complex mistakes
- No PE supports stratified Java
- Design recipes drowned in syntax
abstract class List_Zoo_Animal {
    int fun_for_list();
}

class Cons extends List_Zoo_Animal {
    Zoo_Animal first;
    List_Zoo_Animal rest;

    int fun_for_list() {
        return 1 + rest.fun_for_list();
    }
}

class Empty extends List_Zoo_Animal {
    int fun_for_list() {
        return 0;
    }
}
abstract class List_Zoo_Animal {
    int fun_for_list();
}

class Cons extends List_Zoo_Animal {
    Zoo_Animal first;
    List_Zoo_Animal rest;

    int fun_for_list() {
        return 1 + rest.fun_for_list();
    }
}

class Empty extends List_Zoo_Animal {
    int fun_for_list() {
        return 0;
    }
}
abstract class List_Zoo_Animal {
    int fun_for_list();
}

class Cons extends List_Zoo_Animal {
    Zoo_Animal first;
    List_Zoo_Animal rest;

    int fun_for_list() {
        return 1 + rest.fun_for_list();
    }
}

class Empty extends List_Zoo_Animal {
    int fun_for_list() {
        return 0;
    }
}

This doesn’t include the code needed to actually run the program!
Part V:
Experiences
Sample Exercise

File systems by iterative refinement

#1:
Sample Exercise

File systems by iterative refinement

#1:

A file-or-directory is either
- a file, or
- a directory

A directory is either
- empty
- (cons file-or-directory directory)

A file is a symbol
Sample Exercise

File systems by iterative refinement

#2: A directory is a structure

(make-dir symbol list-of-file/dir)

A file-or-directory is either
• a file, or
• a directory

A file is a symbol

A list-of-file/dir is either
• empty
• (cons file-or-directory list-of-file/dir)
Sample Exercise

File systems by iterative refinement

#3:

A directory is a structure
(make-dir symbol list-of-file/dir)

A file-or-directory is either
• a file, or
• a directory

A file is a structure
(make-file symbol number list-of-values)

A list-of-file/dir is either
• empty
• (cons file-or-directory list-of-file/dir)
Sample Exercise

The functions:
- number-of-files
- disk-usage
- tree-of-disk-usage
- find-file
- all-file-and-directory-names
- empty-directories
- ...
Sample Exercise

File systems by iterative refinement

#1:

A file-or-directory is either
• a file, or
• a directory

A directory is either
• empty
• (cons file-or-directory directory)

A file is a symbol
Sample Exercise

File systems by iterative refinement

#1:

A file-or-directory is either
• a file, or
• a directory

A directory is either
• empty
• (cons file-or-directory directory)

A file is a symbol
File systems by iterative refinement

#1:

A file-or-directory is either
- a file, or
- a directory

A directory is either
- empty
- (cons file-or-directory directory)

A file is a symbol
A file is a symbol

A file-or-directory is either
• a file, or
• a directory

A directory is either
• empty
• (cons file-or-directory directory)

A file is a symbol
File systems by iterative refinement

#3:

A directory is a structure
(make-dir symbol list-of-file/dir)

A file is a structure
(make-file symbol number
list-of-values)

A list-of-file/dir is either
• empty
• (cons file-or-directory list-of-file/dir)

A file-or-directory is either
• a file, or
• a directory
A file is a structure
(make-file symbol number list-of-values)

A file-or-directory is either
• a file, or
• a directory

A directory is a structure
(make-dir symbol list-of-file/dir)

A list-of-file/dir is either
• empty
• (cons file-or-directory list-of-file/dir)
Sample Exercise
Sample Exercise

• Most students are helpless without the design recipe
Sample Exercise

• Most students are helpless without the design recipe
• The templates provide the basic structure of solutions
Sample Exercise

- Most students are helpless without the design recipe
- The templates provide the basic structure of solutions
- The final programs are < 20 lines of actual code
Sample Exercise

- Most students are helpless without the design recipe
- The templates provide the basic structure of solutions
- The final programs are < 20 lines of actual code
- With Teachpack, runs on file system
Sample Exercise

- Most students are helpless without the design recipe
- The templates provide the basic structure of solutions
- The final programs are < 20 lines of actual code
- With Teachpack, runs on file system
- Second midterm (7th/8th week)
Sample Exercise

• Most students are helpless without the design recipe
• The templates provide the basic structure of solutions
• The final programs are < 20 lines of actual code
• With Teachpack, runs on file system
• Second midterm (7th/8th week)
• Exercise extends further (links, ...)
Experiences: Rice University Constraints

- All incoming students
- Life-long learners
- Accommodate industry long-term
- Work after two semesters
Experiences: Rice University Constraints

- All incoming students
- Level playing field, make 1st sem. useful
- Life-long learners
- Accommodate industry long-term
- Work after two semesters
Experiences: Rice University Constraints

- All incoming students
- Life-long learners
- Accommodate industry long-term
- Work after two semesters
- Level playing field, make 1st sem. useful
- Minimize fashions
Experiences: Rice University Constraints

- All incoming students
- Life-long learners
- Accommodate industry long-term
- Work after two semesters

- Level playing field, make 1st sem. useful
- Minimize fashions
- OO, components, etc.
Experiences: Rice University Constraints

- All incoming students
- Life-long learners
- Accommodate industry long-term
- Work after two semesters
- Level playing field, make 1st sem. useful
- Minimize fashions
- OO, components, etc.
- C++ to Java, true OOP
Experiences: The Rice Experiment
Experiences: The Rice Experiment

beginners: none to three years of experience
Experiences: The Rice Experiment

beginners: none to three years of experience
Experiences: The Rice Experiment

comp sci introduction:
- TeachScheme curriculum
- good evaluations
- huge growth
- many different teachers

beginners: none to three years of experience
Experiences: The Rice Experiment

- **comp sci introduction:**
  - TeachScheme curriculum
  - good evaluations
  - huge growth
  - many different teachers

- **applied comp introduction:**
  - C/C++ curriculum
  - weak evaluations
  - little growth
  - several teachers

beginners: none to three years of experience
Experiences: The Rice Experiment

comp sci introduction:
- TeachScheme curriculum
- good evaluations
- huge growth
- many different teachers

applied comp introduction:
- C/C++ curriculum
- weak evaluations
- little growth
- several teachers

beginners: none to three years of experience
Experiences: The Rice Experiment

- comp sci introduction:
  - TeachScheme curriculum
  - good evaluations
  - huge growth
  - many different teachers

- applied comp introduction:
  - C/C++ curriculum
  - weak evaluations
  - little growth
  - several teachers

beginners: none to three years of experience
Experiences: The Rice Experiment

beginners: none to three years of experience

comp sci introduction:
- TeachScheme curriculum
- good evaluations
- huge growth
- many different teachers

applied comp introduction:
- C/C++ curriculum
- weak evaluations
- little growth
- several teachers
Experiences: The Rice Experiment

second semester: OOP, classical data structures, patterns

comp sci introduction:
- TeachScheme curriculum
- good evaluations
- huge growth
- many different teachers

applied comp introduction:
- C/C++ curriculum
- weak evaluations
- little growth
- several teachers

beginners: none to three years of experience
Experiences: The Rice Experiment
Experiences: The Rice Experiment

• Even faculty who prefer C/C++/Java
  – find students from Scheme introduction perform better in 2nd course
  – now teach the Scheme introduction
Experiences: The Rice Experiment

• Even faculty who prefer C/C++/Java
  – find students from Scheme introduction perform better in 2nd course
  – now teach the Scheme introduction

• Students with prior experience eventually understand how much the course adds to their basis
Experiences: The Rice Experiment

- Even faculty who prefer C/C++/Java
  - find students from Scheme introduction perform better in 2nd course
  - now teach the Scheme introduction
- Students with prior experience eventually understand how much the course adds to their basis
- Nearly half the Rice campus takes it!
Experiences: Other Institutions
Experiences: Other Institutions

- Trained nearly 200 teachers/professors
Experiences: Other Institutions

• Trained nearly 200 teachers/professors
• Over 100 deployed and reuse it
Experiences: Other Institutions

• Trained nearly 200 teachers/professors
• Over 100 deployed and reuse it
• Better basis for second courses
Experiences: Other Institutions

• Trained nearly 200 teachers/professors
• Over 100 deployed and reuse it
• Better basis for second courses
• Provides grading rubric
Experiences: Other Institutions

- Trained nearly 200 teachers/professors
- Over 100 deployed and reuse it
- Better basis for second courses
- Provides grading rubric
- Immense help to algebra teachers
Experiences: Other Institutions

- Trained nearly 200 teachers/professors
- Over 100 deployed and reuse it
- Better basis for second courses
- Provides grading rubric
- Immense help to algebra teachers
- Much higher retention rate
  - especially among females
Conclusion
Conclusion

• Training good programmers does not mean starting them on currently fashionable languages and tools
Conclusion

• Training good programmers does not mean starting them on currently fashionable languages and tools

• Provide a strong, rigorous foundation
  – data-oriented thinking
  – value-oriented programming
Conclusion

• Training good programmers does *not* mean starting them on currently fashionable languages and tools

• Provide a strong, rigorous foundation
  – data-oriented thinking
  – value-oriented programming

• *Then, and only then*, expose to i/o details, current fashions, etc.
Conclusion
Conclusion

• Training takes more than teaching some syntax and good examples
Conclusion

- Training takes more than teaching some syntax and good examples

- We must present students with
  - a simple, stratified language
  - an enforcing programming environment
  - a rational design recipe
• Training takes more than teaching some syntax and good examples

• We must present students with
  – a simple, stratified language
  – an enforcing programming environment
  – a rational design recipe

• Teach Scheme!
What We Offer

- **Textbook**: published by MIT Press, available on-line
What We Offer

• **Textbook**: published by MIT Press, available on-line

• Problem sets and solutions
What We Offer

- **Textbook**: published by MIT Press, available on-line
- Problem sets and solutions
- Teacher’s guide, environment guide
What We Offer

- **Textbook**: published by MIT Press, available on-line
- Problem sets and solutions
- Teacher’s guide, environment guide
- **Programming environment**
What We Offer

- **Textbook**: published by MIT Press, available on-line
- Problem sets and solutions
- Teacher’s guide, environment guide
- **Programming environment**
- Teaching libraries
What We Offer

- **Textbook**: published by MIT Press, available on-line
- Problem sets and solutions
- Teacher’s guide, environment guide
- Programming environment
- Teaching libraries
- Summer workshops
What We Offer

- **Textbook**: published by MIT Press, available on-line
- Problem sets and solutions
- Teacher’s guide, environment guide
- Programming environment
- Teaching libraries
- **Summer workshops**

All other than paper book are free
Primary Project Personnel

- Matthias Felleisen Northeastern University
- Matthew Flatt University of Utah
- Robert Bruce Findler University of Chicago
- Shriram Krishnamurthi Brown University

with

- Steve Bloch Adelphi University
- Kathi Fisler WPI

http://www.teach-scheme.org/