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:

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

inner-class declaration
In Pascal/Java/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)))]))

argument interpreted as application

---:-- len2.ss  (Scheme Fill)--L5--All----------------------------------
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
length1 : list -> number
(define (length1 alox)
  (cond
    [(empty? alox) 0]
    [else 1 + (length1 (rest alox))])))

implicit sequencing

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!

• 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 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

• 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

• **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

- **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
• 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

- 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
Program Design for Beginners

- Implicitly foster basic good habits
- Rational in its design
  - its steps explain the code’s structure
- Accessible to beginner
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!
• 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)
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
- 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))
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) … )

is it conditionally defined?
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
    [ (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) … ]
    [ (cons? a-loZA) … (first a-loZA) …
      … (rest a-loZA) … ]))

is the class definition self-referential?
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 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)) … ]))

;; 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)) … ]))

;; 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))) ]))
• 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

• 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.

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

Special case: not reusable!
Factor out commonalities in

• contracts
  – corresponds to parametric polymorphism

• function bodies
  – leads to inheritance and overriding
Design Recipes: Conclusion

- 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?”
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()**
• 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)`
;; 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

- 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?

- Complex notation, complex mistakes
- No PE supports stratified Java
- Design recipes drown 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;
    }
}

This doesn’t include the code needed to actually run the program!
Part V: Experiences
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
A file-or-directory is either
• a file, or
• a directory

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

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 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
Sample Exercise

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
• 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

• Level playing field, make 1st sem. useful

• Life-long learners

• Minimize fashions

• Accommodate industry long-term

• OO, components, etc.

• Work after two semesters

• C++ to Java, true OOP
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

• **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

• 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

• 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

• 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
- 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/