An Object-Oriented World

David Van Horn
Background & Motivation
### The first year

<table>
<thead>
<tr>
<th></th>
<th>Fall</th>
<th>Discrete</th>
<th>Fundies I</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spring</td>
<td>Logic &amp; Comp</td>
<td>Fundies II</td>
<td></td>
</tr>
</tbody>
</table>

Industrial co-op
The first year

<table>
<thead>
<tr>
<th></th>
<th>Fall</th>
<th>Spring</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Discrete</td>
<td>Logic &amp; Comp</td>
<td>Fundies I</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Fundies II</td>
</tr>
</tbody>
</table>

Industrial co-op
The first year

How to Design Programs
An Introduction to Programming and Computing

Matthias Felleisen
Robert Bruce Findler
Matthew Flatt
Shriram Krishnamurthi

;; fact : nat -> nat
(check-expect (fact 0) 1)
(check-expect (fact 5) 120)
(define (fact n)
  (cond [(zero? n) 1]
        [else (* n (fact (sub1 n)))]))

Language: Beginning Student; memory limit: 1024 MB.
Both tests passed!
The first day

#lang htdp/bsl
(require 2htdp/image)
(require 2htdp/universe)

; Use the rocket key to insert the rocket here.

(define ROCKET ?)
(define WIDTH 100)
(define HEIGHT 300)
(define WT-SCENE (empty-scene WIDTH HEIGHT))

; A World is a Number.
; Interp: distance from the ground in AU.
; render : World -> Scene
(check-expect (render 0)
  (place-image ROCKET (/ WIDTH 2) HEIGHT WT-SCENE))

(define (render h)
  (place-image ROCKET
               (/ WIDTH 2)
               (- HEIGHT h)
               WT-SCENE))

; next : World -> World
(check-expect (next 0) 7)
(define (next h)
  (+ h 7))

(big-bang 0
  (on-tick next)
  (to-draw render))
The first day

How to Design Classes
Data: Structure and Organization

Matthias Felleisen
Matthew Flatt
Robert Bruce Findler
Kathryn E. Gray
Shriram Krishnamurthi
Viera K. Proulx
Designing with Class
The first day
The first day
The first day
The next day
The next day
The next day
The next day
The first year

- Inheritance
- Interfaces
- Distributed programming
- Delegation
- Abstraction
- Invariants
- Unit testing
- Random testing
- Types

- Mixins
- Overriding
- Visitors
- Mutation
- Equality
- Implementing OO
- Java
- Generics
- Ruby
- Artificial intelligence

...apologies to John Woo
The first year

FUNDING II (HONORS)
INTRODUCTION TO CLASS-BASED PROGRAM DESIGN

Spring, 2011

The course studies the class-based program design and the design of abstractions that support the design of reusable software and libraries. It covers the principles of object-oriented program design, the basic rules of program evaluation, and examines the relationship between algorithms and data structures, as well as basic techniques for analyzing algorithm complexity.

The course is suitable for both CS majors and non-majors. It assumes that student has been introduced to the basic principles of program design and computation.

Prerequisites

"Think first, experiment later."

The course assumes proficiency with the systematic design of programs and some mathematical maturity. It demands curiosity and self-driven exploration and requires a serious commitment to practical hands-on programming.
4.7 Representing the snake

(all-but-last (field segs)))

This relies on a helper function, all-but-last, which is straightforward to write (recall that segs is a non-empty list):

(check-expect (all-but-last (list "x")) empty)
(check-expect (all-but-last (list "y" "x")) (list "y"))

; (cons x (listof x)) -> (listof x)
; Drop the last element of the given list.
(define (all-but-last ls)
  (cond [(empty? (rest ls)) empty]
        [else (cons (first ls)
                    (all-but-last (rest ls)))]))

The grow method is much like move, except that no element is dropped from the segments list:

(check-expect (send (new snake% "right" (list (new seg% 0 0)) grow)
                   (new snake% "right" (list (new seg% 1 0)
                                           (new seg% 0 0)))))

(define/public (grow)
  (new snake%
    (field dir)
      (cons (send (first (field segs)) move (field dir))
            (field segs))))

Now let's write the turn method:

(check-expect (send (new snake% "left" (list (new seg% 0 0))) turn "up")
              (new snake% "up" (list (new seg% 0 0)))))

(define/public (turn d)
  (new snake% d (field segs)))

And finally, draw:

(check-expect (send (new snake% "left" (list (new seg% 0 0))) draw MT-SCENE)
              (send (new seg% 0 0) draw MT-SCENE))

(define/public (draw scn)
  (fold1 (λ (s scn) (send s draw scn))
        (new snake% 0 0)))
The first year
The first year
The first year

4.7 Representing data structures

This relies on a helper function, all-but-last, which is straightforward (recall that segs is a non-empty list):

```racket
(check-expect (all-but-last (field segs)))
```

The grow method is much like move, except it segments lists:

```racket
(check-expect (send (new snakeX (field dir)) (cons (send (first (field segs))) (new snakeX 'right)))
```

Now let's write the turn method:

```racket
(define/public (turn d) (new snakeX (field segs)))
```

And finally, draw:

```racket
(define/public (draw scn) (foldl (λ (s scn) (send s draw scn)) (field segs))
```

The system for representational strategies has a potentially wide application. For example, you can

1. use it to write a simple interpreter;
2. write a more sophisticated interpreter that can handle bugs and other higher-level features;
3. write a compiler or compiler generator.

- Read and execute the system description. The strategy is parameterized by the.
- For example, to use the strategy for the simple interpreter:
- Use the strategy for the more sophisticated interpreter:
- Use the strategy for the compiler generator.
The first year
Magic Eight Ball
The next years

Bigger data designs

A good story for constructors

Better error messages

Types in class34

Whalesong?
Thanks!

http://www.ccs.neu.edu/course/cs2510h/
{dvanhorn,samth}@ccs.neu.edu