Programming Languages and Techniques (CIS120)

Recap Lecture

April 30, 2014

```
public class Block {
    public static int x;
    public static int y;
    public Block(int x0, int y0) {
        x = x0;
        y = y0;
    }
}

…


public static void main(String[] args) {
  List<Block> list = new LinkedList<Block>();
  list.add(new Block(1,2));
  list.add(new Block(3,4));
  for (Block b : list) {
     System.out.println("x=" + b.x + " y=" + b.y );
  }
}
```

What is printed?

1. x=1 y=2
   x=3 y=4
2. x=3 y=4
   x=3 y=4
3. x=2 y=2
   x=4 y=4
4. NullPointerException

How is HW10 going?

1. not started
2. developing ideas
3. started coding
4. nearly there
5. submitted

Game Project

- Due tonight at midnight
- **No late submissions**

- Schedule a demo session with your TA
- See assignment webpage for grading rubric. Be prepared to discuss your game at the demo.

- TAs will continue OH until the exam where possible, but will have to reshuffle based on their own exam schedules

FINAL EXAM

- **Wednesday May 7, 9-11 AM**
  - DRLB A1, Last name A-N
    DRLB A8, Last name P-Z

- Comprehensive exam covering *all course content*:
  - both OCaml and Java
  - Concepts from homework assignments and lectures

- Closed book
  - One letter-sized, handwritten sheet of notes allowed

- TA-led Review
  - 6-9PM Saturday, Levine 101 (pizza!)
- Mock Exam:
  - 6-9PM Sunday, Levine 101 (6-8 proctored Spring 2012 final, 8-9 review)
- Review material posted on course web page

What did you think of the use of clickers this semester?

1. worked well – definitely keep using them
2. no strong opinion
3. didn’t like it

How often did you watch the lecture screencasts?

1. Frequently, to review concepts
2. Sometimes, to replay tricky concepts
3. Sometimes, to make up missed lectures
4. Rarely
5. There are screencasts available?

CIS 120 Recap

13 concepts in 37 lectures

Concept: Design Recipe

1. Understand the problem
   What are the relevant concepts and how do they relate?
2. Formalize the interface
   How should the program interact with its environment?
3. Write test cases
   How does the program behave on typical inputs? On unusual ones? On erroneous ones?
4. Implement the required behavior
   Often by decomposing the problem into simpler ones and applying the same recipe to each

Did you find writing unit tests useful?

1. yes, it helped me to understand the problem before I started coding
2. yes, it helped me to help me debug while coding
3. yes, it helped me to be sure my homework was correct before I submitted it
4. no, I wrote tests only because I wanted to get full credit on the assignments
5. no, I never wrote tests

Test Driven Development

- Concept: Write tests *before* coding
  - *"test first"* methodology

- Examples:
  - Simple assertions for declarative programs (or subprograms)
  - Longer (and more) tests for stateful programs / subprograms
  - Informal tests for GUIs (can be automated through tools)

- Why?
  - Tests clarify the specification of the problem
  - Thinking about tests informs the implementation
  - Tests help with extending and refactoring code later
    - automatic check that things are not getting broken

Persistent data structures

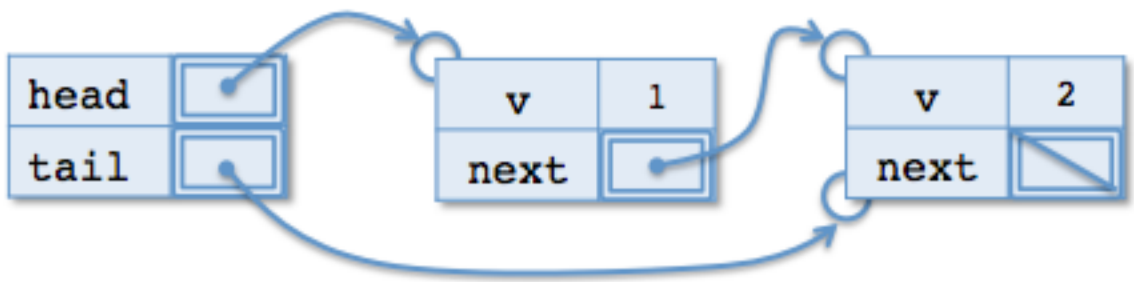
- Concept: Store data in *persistent*, *im*... implement computation as *tra*... structures
- Examples: immutable lists and ... Pictures and Strings in Java (HW...
- Why?
  - Simple model of com..., si...
  - Simple interface: ... have to reas... communication ...etween various parts of the program, all interfaces are explicit)
  - *Recursion* amenable to mathematical analysis (CIS 160/121)
  - Plays well with parallelism

Recursion is the natural way of computing a function f(t) when t belongs to an inductive data type:

1. Determine the value of f for the base case(s).
2. Compute f for larger cases by combining the results of recursively calling f on smaller cases.

Mutable data structures

- Concept: Some data structures are *ephemeral*: computations mutate them over time
- Examples: queues, deques (HW5), GUI state (HW6, 8), arrays (HW 7), dynamic arrays, dictionaries (HW9), hashtables, game state (HW 10)



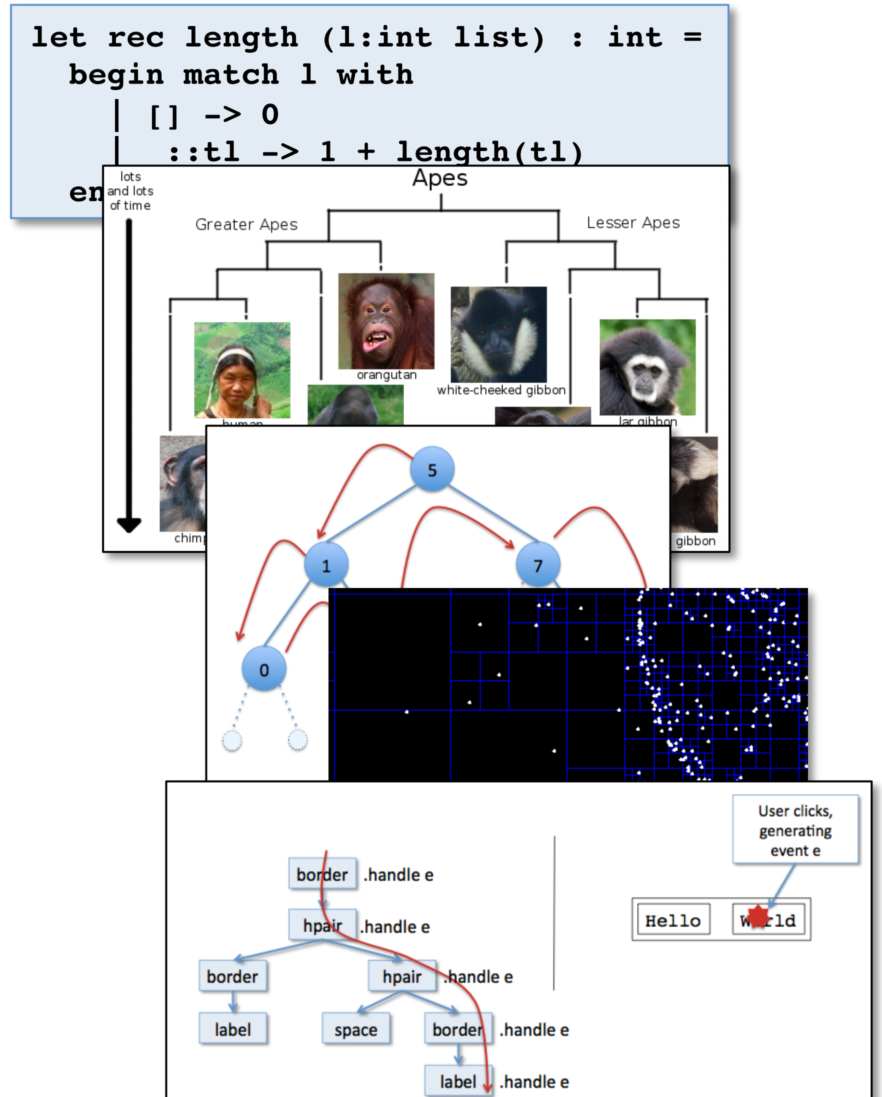
A queue with two elements

- Why?
  - Common in OO programming, which simulates the transformations that objects undergo when interacting with their environment
  - Heavily used for event-based programming, where different parts of the application communicate via shared state
  - Default style for Java libraries (collections, etc.)

Concept: Trees

- Lists (i.e. “unary” trees)
- Simple binary trees
- Trees with invariants: e.g. binary search trees
- Quad trees: spatial search
- Widget trees: screen layout + event routing
- Swing components

- Both *persistent* and *mutable* trees are ubiquitous!



First-class computation

- Concept: code is a form of data that can be defined by functions, methods, or objects (including anonymous ones), stored in data structures, and passed to other functions
- Examples: map, filter, fold (HW4), dynamic dispatch, event listeners (HW6, 8, 10)

```
cell.addMouseListener(new MouseAdapter() {
    public void mouseClicked(MouseEvent e) {
        selectCell(cell);
    }
});
```

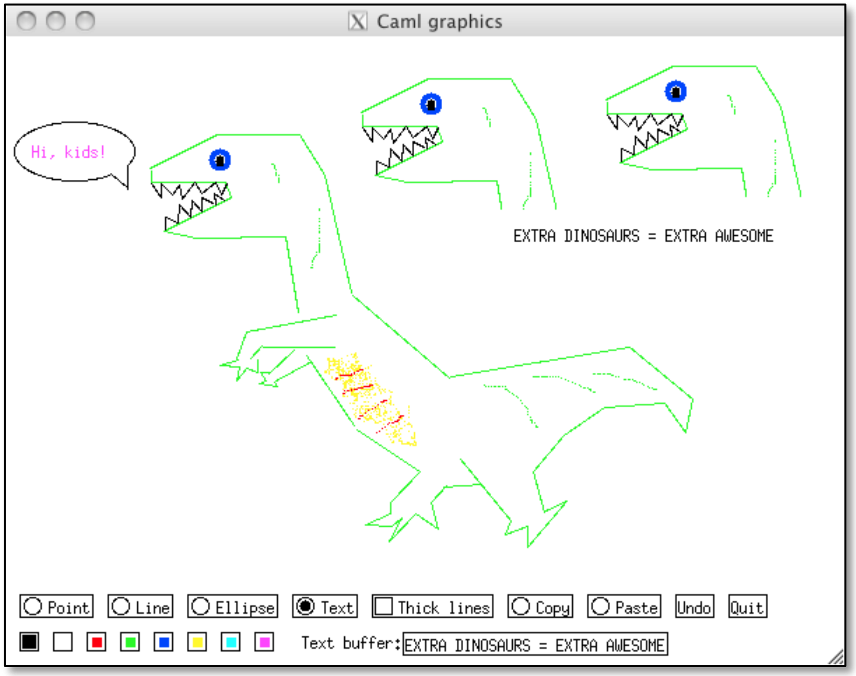
- Why?
  - Powerful tool for abstraction: can factor out design patterns that differ only in certain computations
  - Heavily used for *reactive* programming, where data structures store "reactions" to various events

Event-Driven programming

- Concept: Structure a program by associating "handlers" that run in reaction to program events. Handlers typically interact with the rest of the program by modifying shared state.

- Examples: GUI programming in OCaml and Java

- Why?
  - Practice with reasoning about shared state
  - Practice with first-class functions
  - Necessary for programming with Swing
  - Fun!



Types, Generics, and Subtyping

- Concept: *Static type systems* prevent errors. Every expression has a static type, and OCaml/Java use the types to rule out buggy programs. *Generics* and *subtyping* make types more flexible and allow for better code reuse.
- Examples: entire course

```
let rec contains (x:'a) (l:'a list) : bool =
  begin match l with
    | [] -> false
    | h::tl -> x = a || (contains x tl)
  end
```

- Why?
  - Easier to fix problems indicated by a type error than to write a test case and then figure out why the test case fails
  - Promotes refactoring: type checking ensures that basic invariants about the program are maintained

Abstract types and encapsulation

- Concept: *Type abstraction* hides the actual implementation of a data structure, describes a data structure by its interface, and supports reasoning with invariants

- Examples: Set/Map interface (HW3), queues in OCaml (HW 5) and Java, encapsulation and access control

mentation without modifying clients

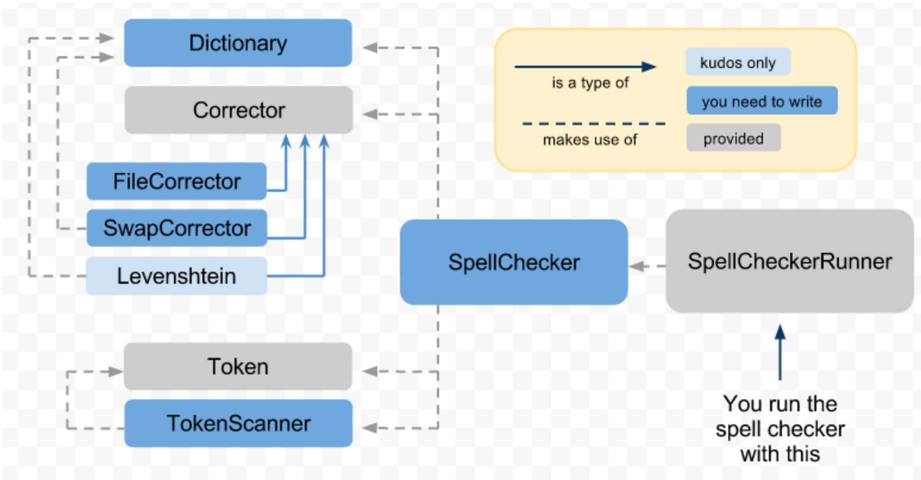
ants about the implementation

Invariants are a crucial tool for reasoning about data structures:

1. *Establish* the invariants when you create the structure.
2. *Preserve* the invariants when you modify the structure.

Sequences, Sets and Finite Maps

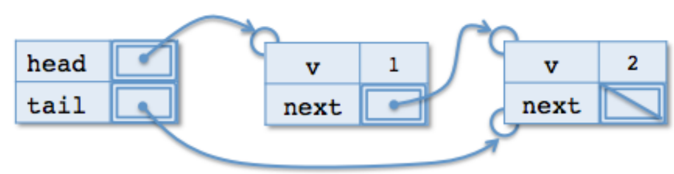
- Concept: semantics of three key *abstract* data structures
- Examples: HW3, Java Collections, Iterators, HW09
- Why?
  - These abstract data types come up again and again
  - Need aggregate data structures (collections) no matter what language you are programming in
  - Need to be able to choose the data structure with the right semantics



Lists, Trees, BSTs, Queues, and Arrays

- Concept: key **implementations** for sequences, sets and finite maps
- Examples: HW2-5, Java Collections, DynamicArrays, Hashtables
- Why?
  - Need some concrete implementation of the abstract types
  - Different implementations have different trade-offs. Need to understand these trade-offs to use them well.
  - For example: BSTs use their invariants to speed up lookup operations compared to linked lists.

interface Set {boolean isEmpty(); …}



A queue with two elements

Lookup 'a'

Lookup 'a'

Lookup 'b'

Lookup 'b'

Do the Function call

Save Workspace; push l1, l2

Lookup l1

Lookup l1

Match Expression

Nil case Doesn't Match

Cons case *Does* Match

Simplify the Branch: push h, t

Lookup 'h'

Lookup 'h'

Lookup 'append'

Lookup 'append'

Did you find the Abstract Stack Machine useful?

1. yes, I never write code without drawing pictures first
2. yes, stepping through the ASM helps me debug
3. yes, it helped me to understand how various features in OCaml and Java work, but I don't use it for implementation
4. no, I don't really understand it
5. no, I don't see the point

Abstract Stack Machine

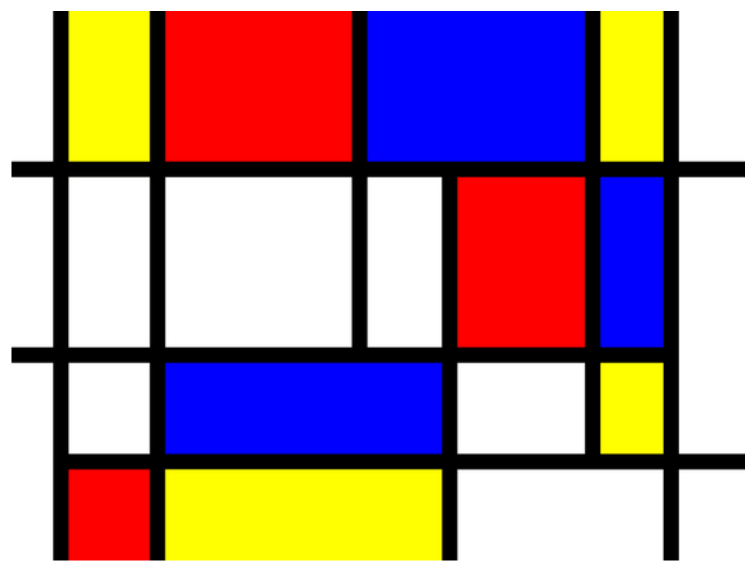
- Concept: The *Abstract Stack Machine* is a detailed model of the execution of OCaml/Java
- Example: throughout the semester!
- Why?
  - To know what your program does without running it
  - To understand tricky features of Java/OCaml language (first-class functions, exceptions, static members, dynamic dispatch, overriding)
  - To help understand the programming models of other languages: Javascript, Python, C++, C#, …

Abstraction

- Concept: *Don't Repeat Yourself!*
  - Find ways to generalize code so it can be reused in multiple situations
  - Simplify interactions between components by hiding details

- Examples: Functions/methods, generics, higher-order functions, interfaces, subtyping, abstract classes

- Why?
  - Duplicated functionality = duplicated bugs
  - Duplicated functionality = more bugs waiting to happen
  - *Good abstractions make code easier to read, modify, maintain and reuse*



Onward…

What Next?

- Classes:
  - CIS 121, 262, 320 – data structures, performance, computational complexity
  - CIS 19x – programming languages and technologies
    - C++, C#, Python, Haskell, Ruby on Rails, iPhone programming
  - CIS 240 – lower-level: hardware, gates, assembly, C programming
  - CIS 341 – compilers (projects in OCaml!)
  - CIS 371, 380 – hardware and OS's
  - CIS 552 – advanced programming
  - And much more!
- Undergraduate research

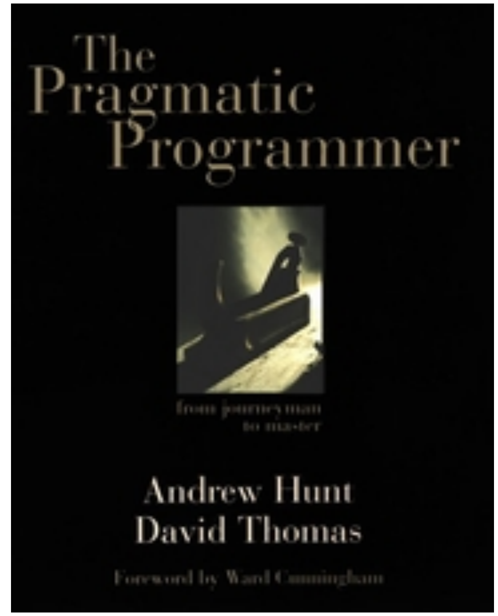
WiCS
women in computer science

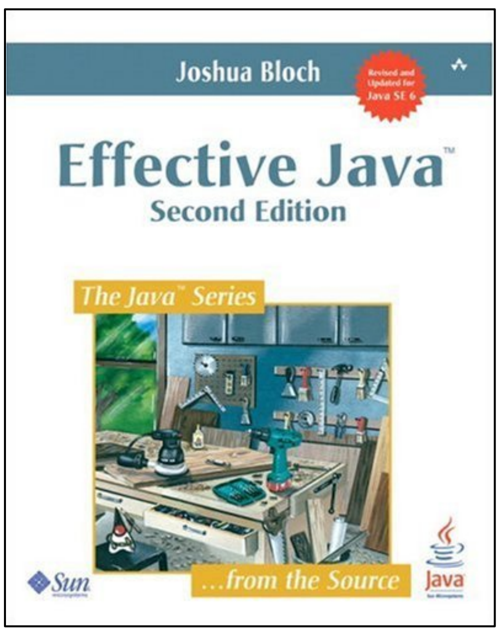
data hackathon sept 16-18
pennapps

dining philosophers
UNIVERSITY OF PENNSYLVANIA COMPUTER SCIENCE CLUB

The Craft of Programming

- *The Pragmatic Programmer:*
  From Journeyman to Master
  by Andrew Hunt and David Thomas
  - Not about a particular programming language, it covers style, effective use of tools, and good practices for developing programs.

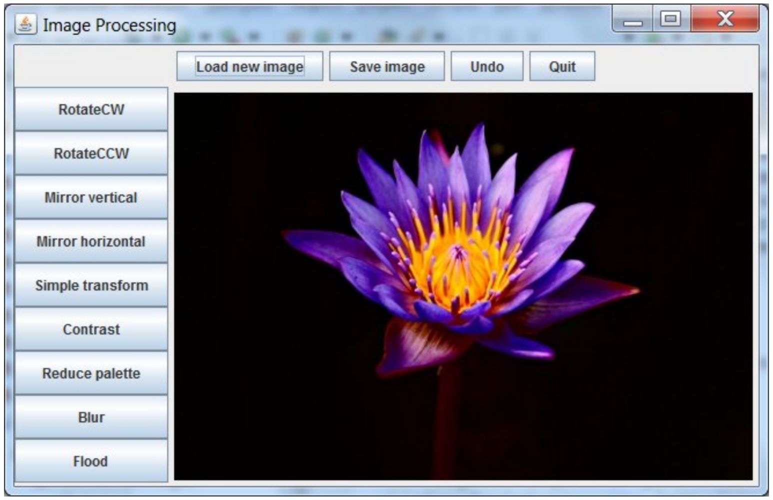




- *Effective Java*
  by Joshua Bloch
  - Technical advice and wisdom about using Java for building software. The views we have espoused in this course share much of the same design philosophy.

Parting Thoughts

- Improve CIS 120:
  - End-of-term survey (will be posted soon on Piazza)
  - Penn Course evaluations also provide useful feedback
  - We take them seriously: please complete them!



Thanks!

```
let rec length (l:int list) : int =
  begin match l with
    | [] -> 0
    | _::tl -> 1 + length(tl)
  end
```