Programming Languages and Techniques (CIS120)

Lecture 37

April 22, 2013

Recap

Game Project

- Due tomorrow at Midnight
- Normal late policy (10 points per day, up to two days)

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

- Weirich OH today and next Monday (3:30-5PM)
- TAs will continue OH, but will have to reshuffle based on their own exam schedules

FINAL EXAM

- **Friday May 3rd, 9-11AM**
  - **[A-L] Towne 100**
  - **[M-T] Skirkanich Auditorium**
  - **[V-Z] Towne 309**

- *Comprehensive* exam over course concepts:
  - OCaml material (though we won't worry much about syntax)
  - Java material
  - all course content

- Closed book
  - One letter-sized, handwritten sheet of notes allowed
- TA Review Session:
  - TBA
  - Review material posted on course web page

Grade database

- Check your scores online for errors
  - Homework 1-8, Midterms 1&2 should be correct
  - Lab attendance, HW 9 grades will be entered by the end of the week

- Send mail to tas120 if you are missing any grades.

- You are looking at the same database I will use to calculate final grades.
  - Homework 50%
  - Labs 6%
  - First midterm 12%
  - Second midterm 12%
  - Final exam 20%

CIS 120 Concepts

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

Unit Testing

- 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 also helps with refactoring (let you know that you haven't broken anything)

Persistent data structures

- Concept: Store data in i… …ment computation as *transf…*
- Examples: immutable … 3), images and Strings in J…
- Why?
  - Simple model of co…
  - Simple interface… …it communicatio… …twee… …aces are explicit)
  - *Recursion* amenable to mathematical analysis (CIS 121)
  - Have all intermediate values available

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

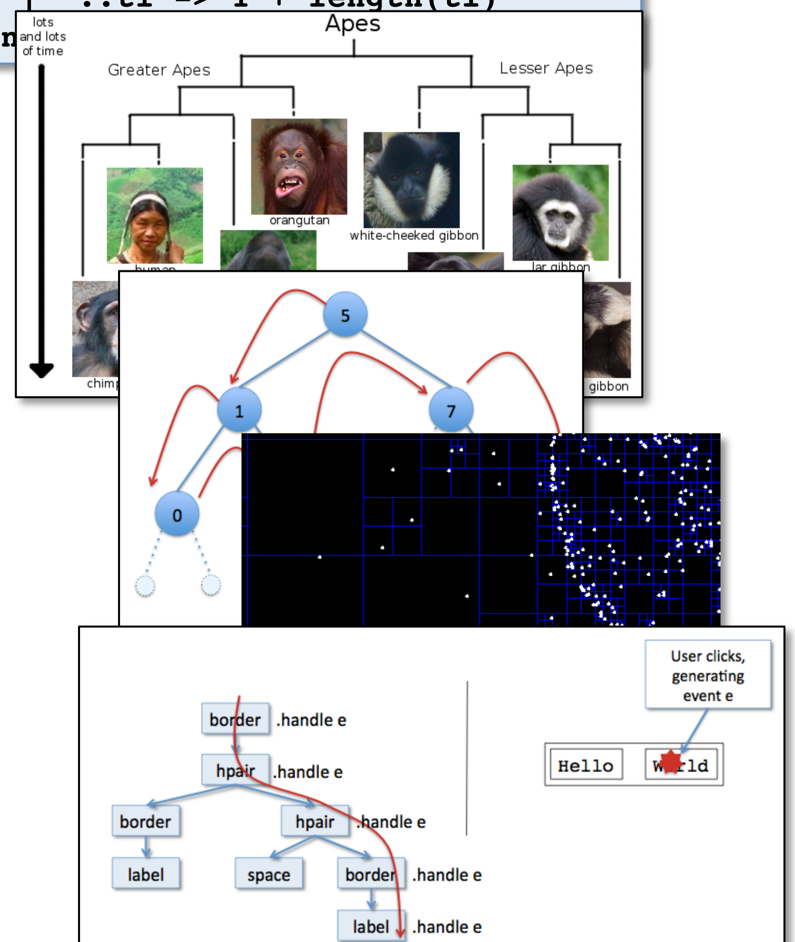
1. Determine the value of f for the base case(s).
2. Show how to compute the value of f for larger cases by combining the results of recursively calling f on smaller cases.

Trees

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

- Trees are ubiquitous in CS:
  - file system folders
  - URLs
  - hierarchy

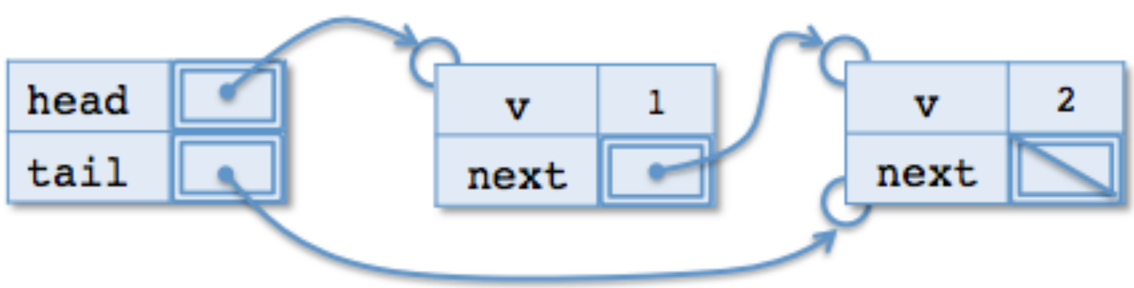
```
let rec length (l:int list) : int =
  begin match l with
    | [] -> 0
    | ::tl -> 1 + length(tl)
  en
```



Mutable data structures

- Concept: Some data structures are ephemeral. Computation based on modifications of those data structures over time.
- Examples: Queues, Deques (HW5), GUI state (HW6), arrays (HW 6), Dynamic Arrays, Characters (HW 8), Dictionaries (HW9)

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



A queue with two elements

First-class computation

- Concept: code is a kind of data that can be defined in functions or methods, stored in data structures, and passed to other functions.
- Examples: map, fold (HW4), pixel transformer (HW7), Event listeners (HW6, HW10)

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

- Why?
  - Allows more flexibility in the structure of code, can factor out design patterns that differ only in certain computations
  - Necessary for reactive programming, where data structures store the "reactions" to various events

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 code reuse.

```
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 datastructure, describes a datastructure by its interface (what it does vs. how it is represented), allows the use of reasoning with invariants
- Examples: Set/Map interface (HW3), Queues in OCaml and Java, encapsulation and access control (HW8)

Invariants are a crucial way of structuring code and data:

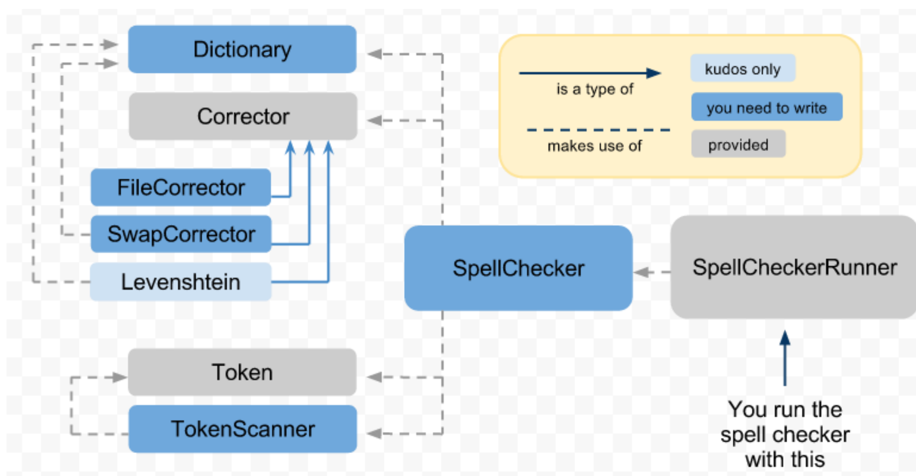
1. *Establish* the invariants when you create the structure.
2. *Preserve* the invariants when you modify the structure.

...ithout modifying clients

...mentation *invariants* about the

Sequences, Sets and Finite Maps

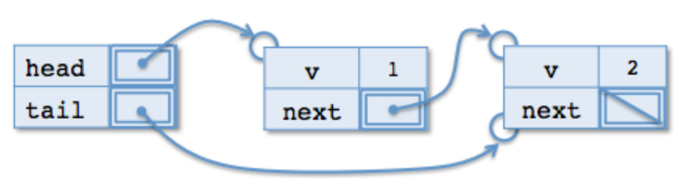
- Specific **abstract** data types: sequences, sets and finite maps
- Examples: HW3, Java Collections, HW09
- Why?
  - These abstract data types come up again and again
  - Need aggregate data structures (collections) no matter what language you are programming in
  - Need to be able to choose the data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



Lists, Trees, BSTs, Queues and Arrays

- Concept: specific implementations for abstract types
- Examples: HW2-5, Java Collections
- 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'

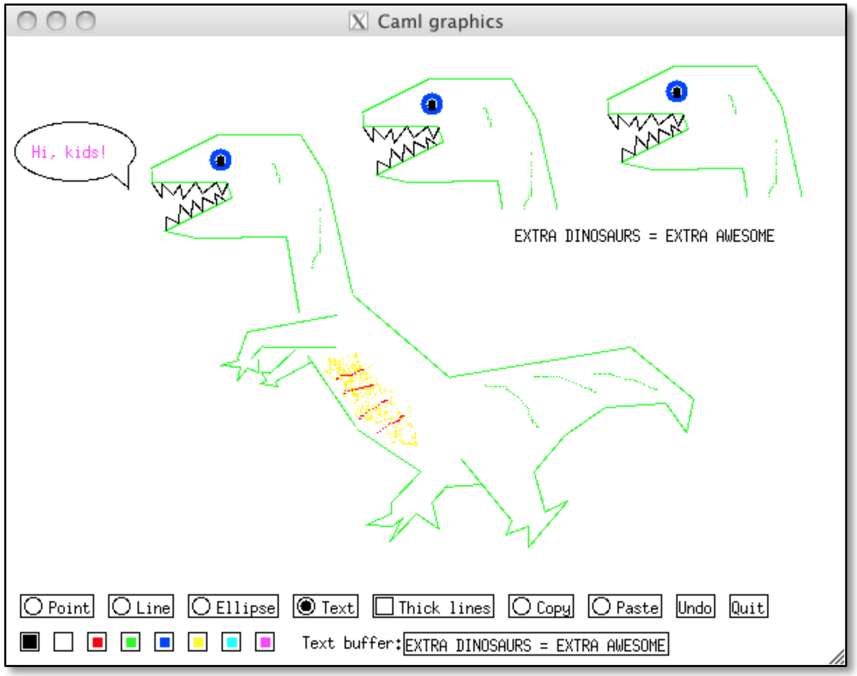
Lookup 'append'

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



Abstraction

- Concept: Don't Repeat Yourself. Find some way to generalize the code that you write to apply to more situations
- Examples: Functions/methods, generics, higher-order functions, interfaces, subtyping, abstract classes

- Why?
  - If you only write your code once, you only have to debug it once
  - Makes code easier to extend, can reuse the same code many times
  - Makes code easier to read, if parts of your program are meant to be similar, you can tell by reading the program

Why OCaml?

Why some other language?

- Experience with learning a new language
- Perspective about language-independent concepts
- Account for varying degrees of experience in the same class

…but, why OCaml?

Rich, orthogonal vocabulary

- In Java, primitive types, arrays, objects
- In OCaml, primitive types, arrays, objects, datatypes (including lists, trees, and options), records, refs and first-class functions and abstract types

- All of the above *can* be implemented in Java, but untangling various use cases of objects is subtle
- Concepts (like generics) can be studied in isolation, fewer intricate interactions with the rest of the language

Functional Programming

- In Java, every reference is mutable and optional by default
- In OCaml, persistent data structures are the default. Furthermore, the type system keeps track of what is and is not mutable, and what is and is not optional

- Advantages of immutable/persistent data structures
  - Don't have to keep track of aliasing. Interface to the data structure is simpler
  - Often easier to think in terms of "transforming" data structures than "modifying" data structures
  - Simpler implementation (Compare lists and trees to queues and deques)
  - Powerful evaluation model (substitution + recursion).

Why Java?

Object Oriented Programming

- Provides a different way of decomposing programs
- Basic principles:
  - Encapsulation of local, mutable state
  - Inheritance to share code
  - Dynamic dispatch to select which code gets run

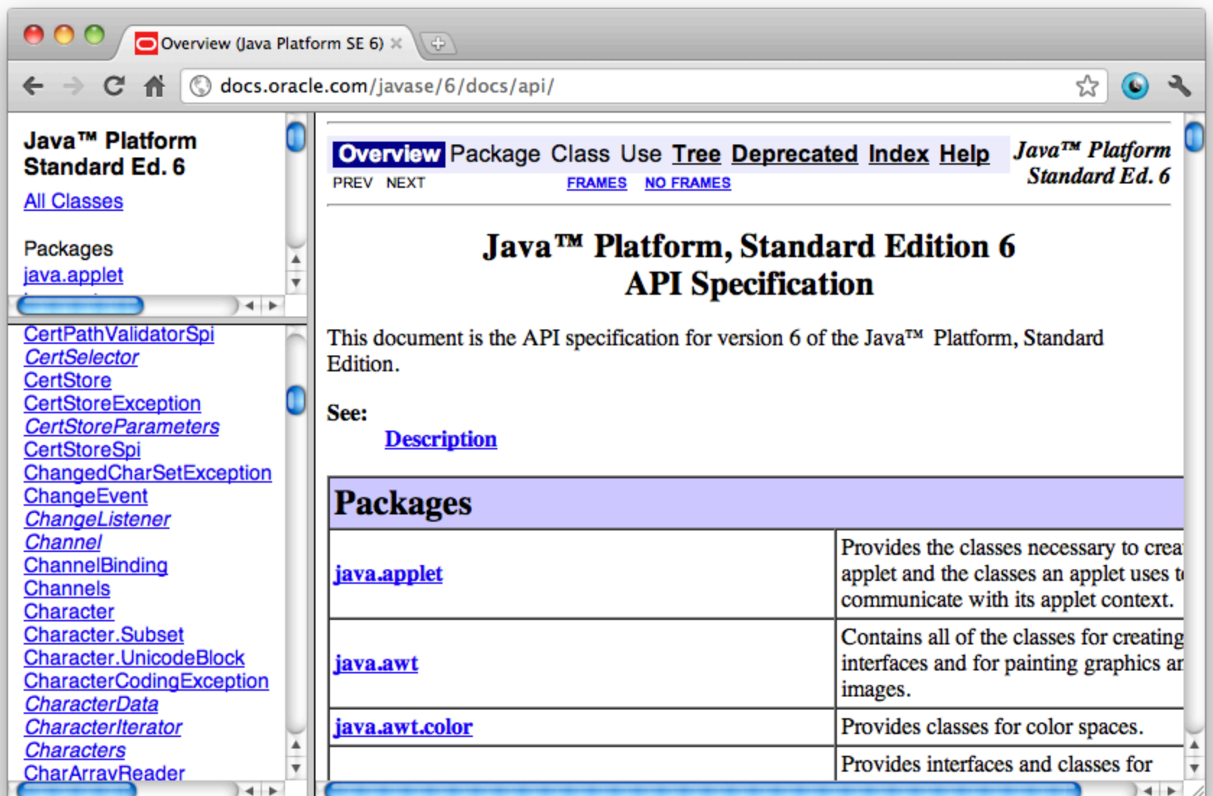
```
Welcome to the Adventure Game.
Type "help" at any time to get a list of available commands.

You are in the ballroom.
There are exits to the south and east.
You have 10 health and 7 coins.
>>>
```

- but why Java?

“Real” Programming Ecosystem

- Industrial strength tools:
  - Eclipse
  - JUnit testing framework
- Libraries:
  - Swing
  - Collections
  - I/O libraries
  - …



Onward…

What Next?

- Classes:
  - CIS 121, 262, 320 – data structures, performance, computational complexity
  - CIS 19x – programming languages
    - 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
  - And much more!
- Undergraduate research

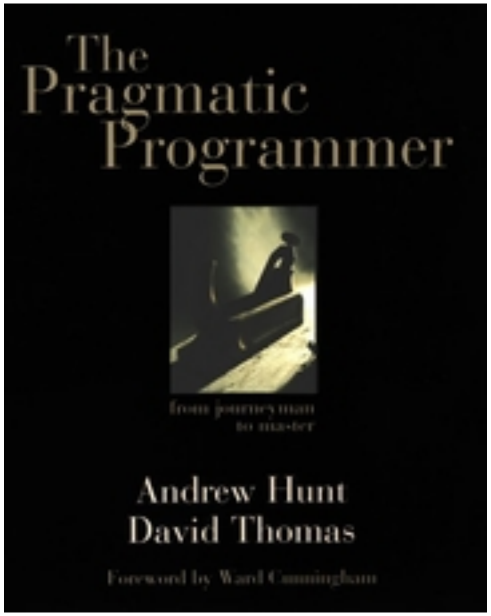
WiCS
women in computer science

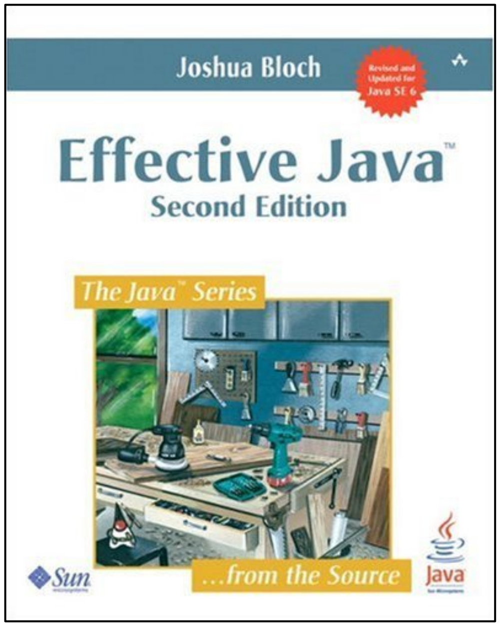
data hackathon sept 16-18
pennapps

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.

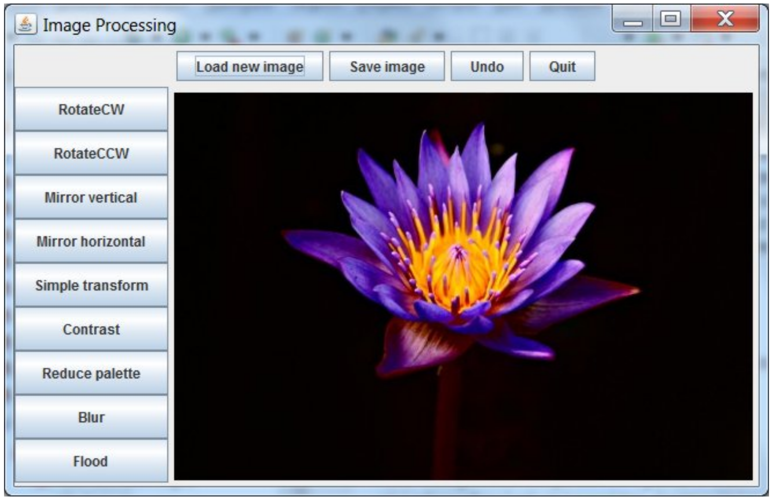




- *Effective Java*
  by Joshua Bloch
  – Technical advice and wisdom about using Java for building software. The views we have espoused in this course share much of the same design philosophy.

Parting Thoughts

- Improve CIS 120:
  - End-of-term survey will be sent soon
  - Penn Course evaluations also provide useful feedback
  - We take them seriously, please complete them!



Finally: Thanks!

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