Programming Languages and Techniques (CIS120)

Lecture 38

April 27, 2016

Semester Recap

FINAL EXAM

- **Monday, May 9th,** 9-11AM
  - LRSM AUD [A - Edson]
  - TOWN 100 [Efremkina - Mendivil]
  - LEVH 101 [Mitchnick - Taitz]
  - SKIR AUD [Tallini - Z]
- *Comprehensive* exam over course concepts:
  - OCaml material (though we won't worry much about syntax)
  - All Java material (emphasizing material since midterm 2)
  - all course content
  - old exams posted
- Closed book, but:
  - One letter-sized, single-sided, *handwritten* sheet of notes allowed

Review Sessions

- Mock Exam
  - Saturday, May 7th
  - 2:00pm – 4:00pm, answers 4-5PM
  - Location: Towne 100

- Review Session
  - Saturday, May 7th
  - 6:00pm – 8:00pm
  - Location: Towne 100

- Office Hours
  - See online Schedule & Piazza

Grade database

- Check your scores online for errors
  - Homework 1-6, Midterms 1&2, class participation (quizzes), lab attendance
  - HW 7, 8 grades will be entered soon!

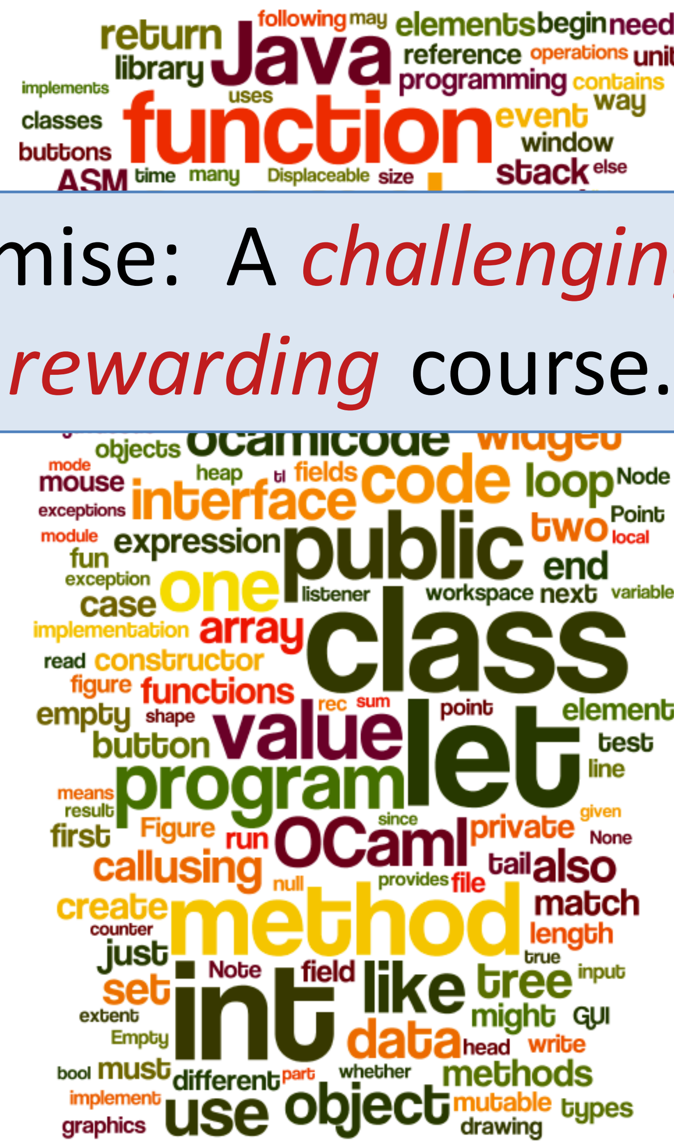
- Send mail to tas120@seas if you are missing any grades

- You are looking at the same database I will use to calculate final grades…
  - Homework 50% (50%/9 per project)
  - Labs 6%
  - First midterm 12%
  - Second midterm 12%
  - Final exam 18%
  - Class participation 2%

CIS 120 Recap

From Day 1

- CIS 120 is a course in **program design**
- Practical skills:
  - ability to write larger (~1000 lines) programs
  - increased independence ("working without a recipe")
  - test-driven development, principled debugging
- Conceptual foundations:
  - common data structures and algorithms
  - several different programming idioms
  - focus on modularity and compositionality
  - derived from first principles throughout
- It will be fun!



Promise: A *challenging* but *rewarding* course.

Which assignment was the most challenging?

1. OCaml finger exercises
2. DNA
3. Sets and Maps
4. Queues
5. GUI
6. Images
7. Chat
8. SpellChecker
9. Game

Which assignment was the most rewarding?

1. OCaml finger exercises
2. DNA
3. Sets and Maps
4. Queues
5. GUI
6. Images
7. Chat
8. SpellChecker
9. Game

CIS 120 Concepts

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?

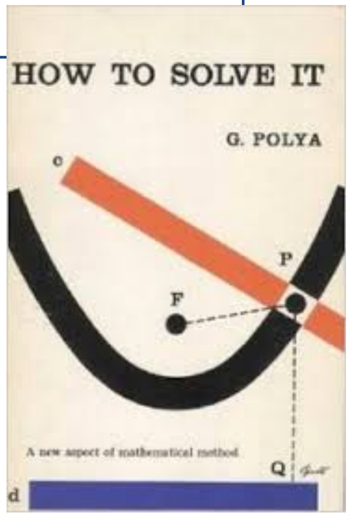
3. Write test cases

   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

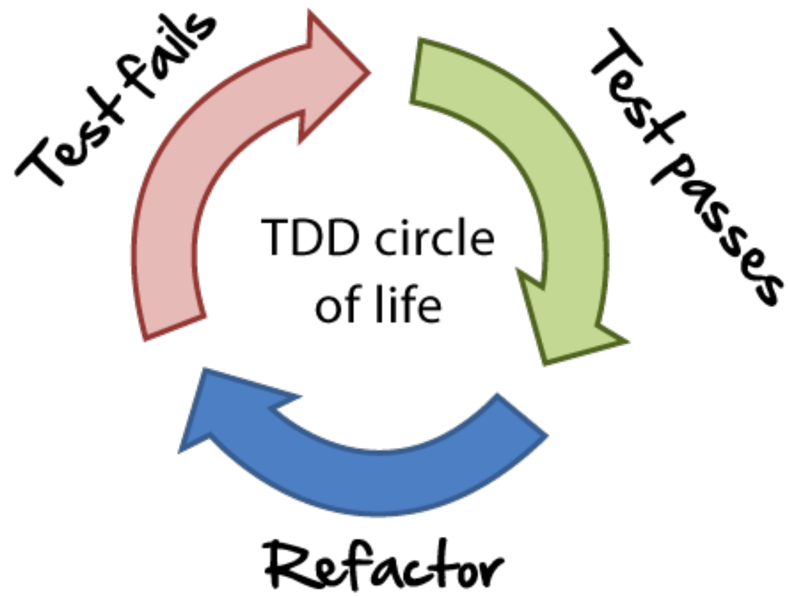
"Solving problems", wrote Polya, "is a practical art, like swimming, or skiing, or playing the piano: You can learn it only by imitation and practice."



Testing

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

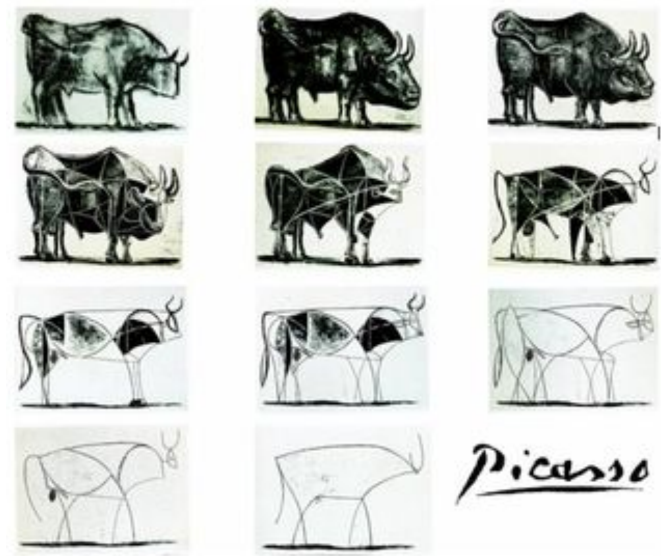
- Examples:
  - Simple assertions for declarative programs (or subprograms)
  - Longer (and more) tests for stateful programs / subprograms
  - Informal tests for GUIs (can be automated through tools)



- Why?
  - Tests clarify the specification of the problem
  - Thinking about tests informs the implementation
  - Tests help with extending and refactoring code later
  - Industry practice

Functional/Procedural Abstraction

- Concept: *Don't Repeat Yourself!*
  - Find ways to generalize code so it can be reused in multiple situations



Pablo Picasso, Bull (plates I - XI) 1945

- 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

Persistent data structures

- Concept: Store data in *persistent, immutable* structures; implement computation as *tra*
  structures

- Examples: immutable lists and
  images and Strings in Java (HW

- Why?
  - Simple model of comp   si
  - Simple interface: D   have to re
    communication b   ween various parts of the program, all interfaces are explicit)
  - *Recursion* amenable to mathematical analysis (CIS 160/121)
  - Plays well with parallelism

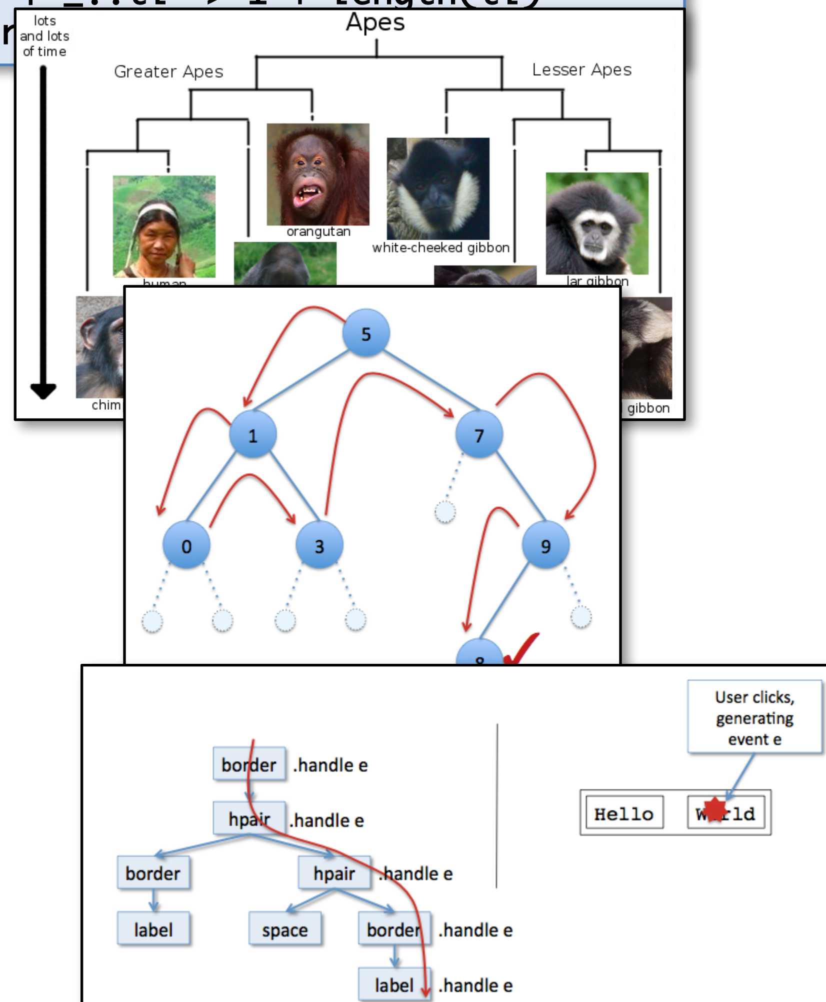
Recursion is the natural way of computing a function f(t) when t belongs to an inductive data type:

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.
3. Same idea as mathematical induction (a la CIS 160)

Concept: Tree Structured data

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

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



- Why? Trees are ubiquitous in CS!
  - file system organization
  - languages, compilers
  - domain name hierarchy www.google.com

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), pixel transformers (HW6), event listeners (HW5, 7, 9)

```
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

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.

```
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 (what it does vs. how it is represented), supports reasoning with invariants

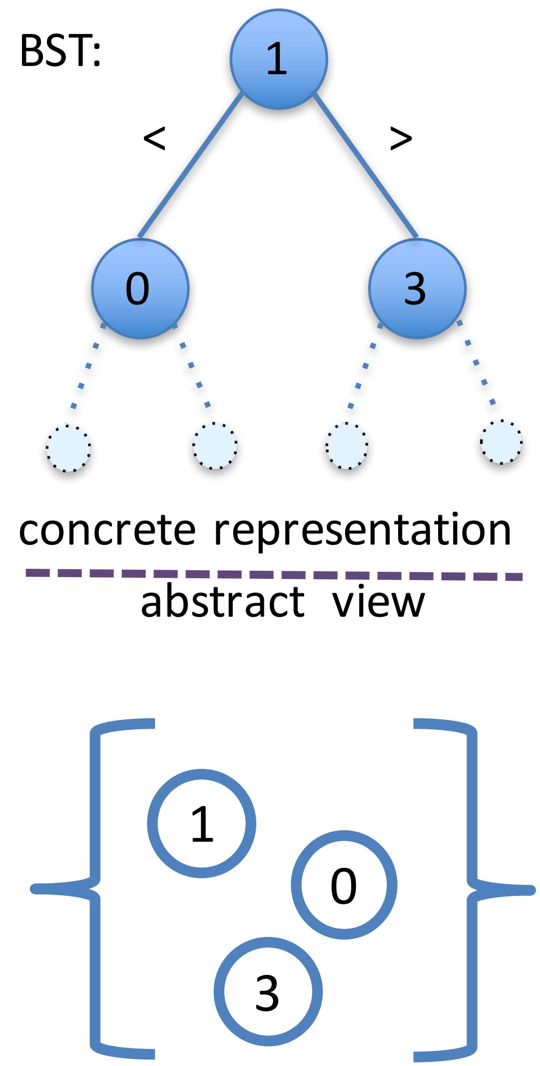
- Examples: Set/Map interface (HW3), queues in
  and access

  entation without

  about the

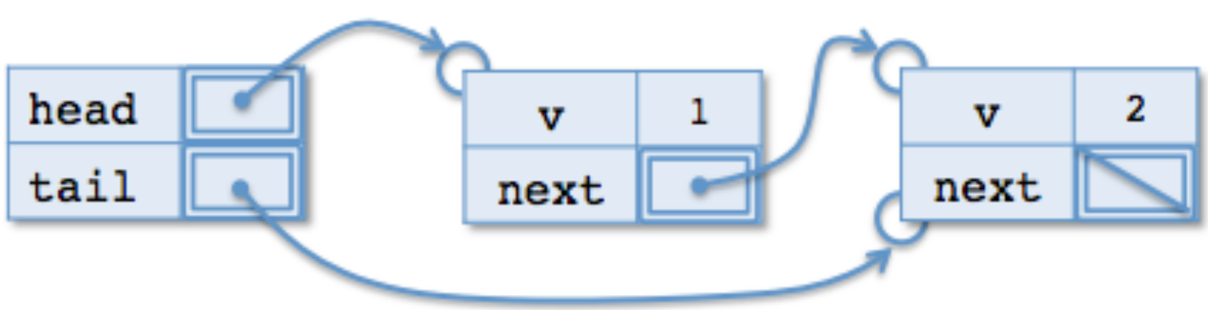
Invariants are a crucial tool for reasoning about data structures:

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



Mutable data

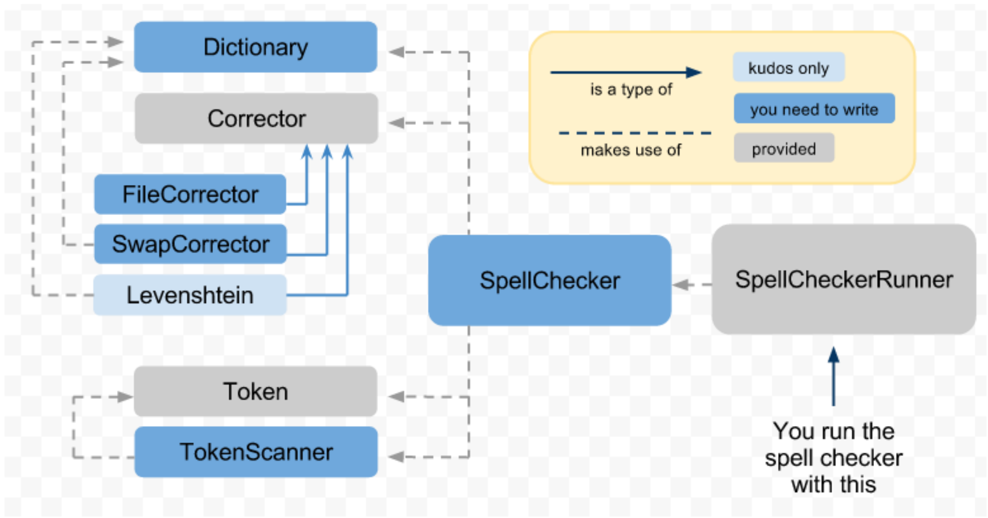
- Concept: Some data structures are *ephemeral*: computations mutate them over time
- Examples: queues, deques (HW4), GUI state (HW5, 9), arrays (HW 6), dynamic arrays, dictionaries (HW8)
- Why?
  - Common in OO programming, which simulates the transformations that objects undergo when interacting with 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.)



A queue with two elements

Sequences, Sets, Maps

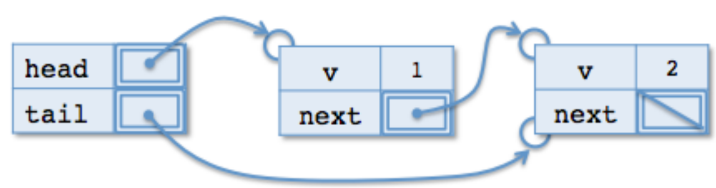
- Concept: Specific **abstract data types** of sequences, sets, and finite maps
- Examples: HW3, Java Collections, HW 7, 8
- Why?
  - These abstract data types come up again and again
  - Need aggregate data structures (collections) no matter what language you are programming in
  - Need to be able to choose the data structure with the right semantics



Lists, Trees, BSTs, Queues, and Arrays

- Concept: There are *implementation trade-offs* for abstract types
- Examples:
  - Binary Search Trees vs. Lists vs. Hashing for sets and maps
  - Linked lists vs. Arrays for sequential data
- Why?
  - Abstract types have multiple implementations
  - 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

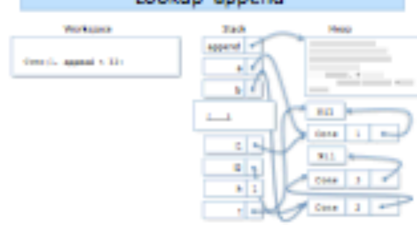
Cons case *Does* Match

Simplify the Branch: push h, t

Lookup ‘h’

Lookup ‘h’

Lookup ‘append’



Lookup ‘append’

Abstract Stack Machine

- Concept: The *Abstract Stack Machine* is a detailed model of the execution of OCaml/Java

- Example: throughout the semester!

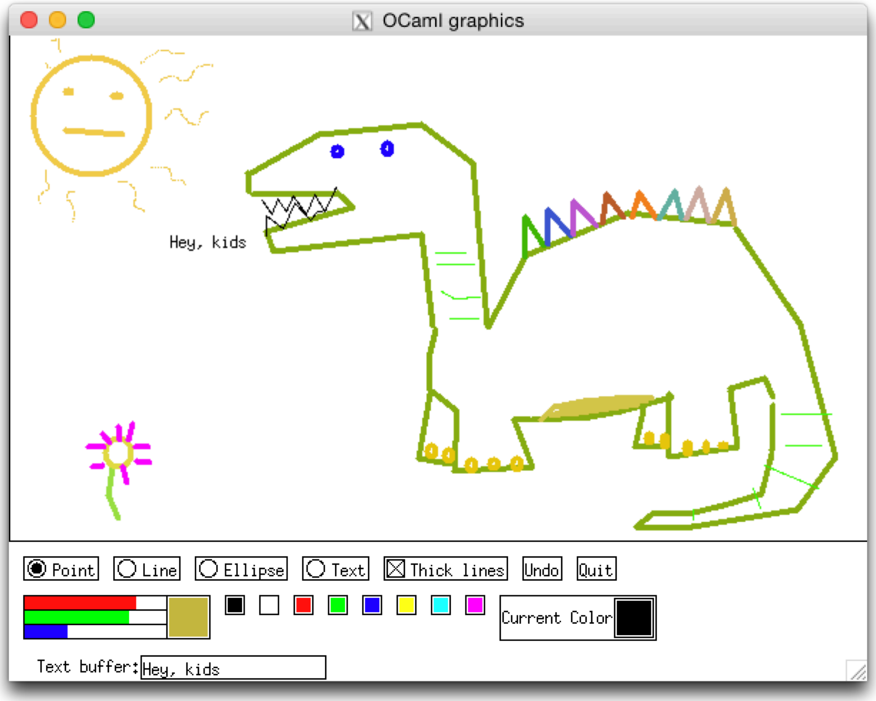
- Why?
  - To know what your program does without running it
  - To understand tricky features of Java/OCaml language (aliasing, first-class functions, exceptions, dynamic dispatch)
  - To help understand the programming models of other languages: Javascript, Python, C++, C#, …
  - To predict performance and space usage behaviors

Event-Driven programming

- Concept: Structure a program by associating "handlers" that *react 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
  - Common in GUI applications



Why OCaml?

Why some other language than Java?

- Level playing field for students with varying backgrounds coming into the same class
- Two points of comparison allow us to emphasize language-independent concepts

…but, why specifically OCaml?

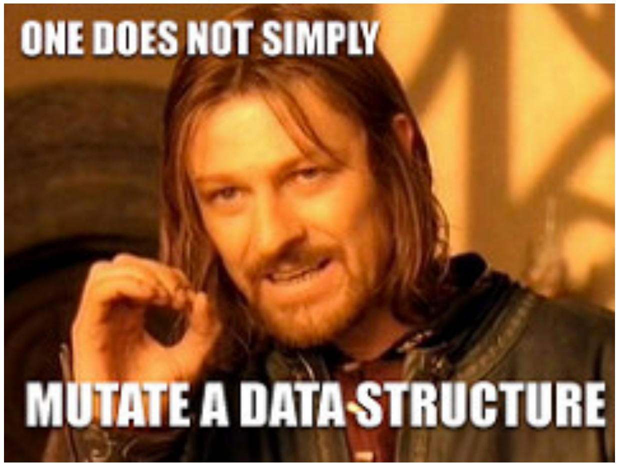
Rich, orthogonal vocabulary

- In Java: int, A[], Object, Interfaces
- In OCaml:
  - primitives
  - arrays
  - objects
  - datatypes (including lists, trees, and options)
  - records
  - refs
  - first-class functions
  - 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).



Who uses OCaml?

intel
facebook
JANE STREET
LexiFi
Google
CITRIX
Microsoft
MLstate
cea
Microsoft Visual F#
mylife
SimCorp
DASSAULT AVIATION

Why Java?

Object Oriented Programming

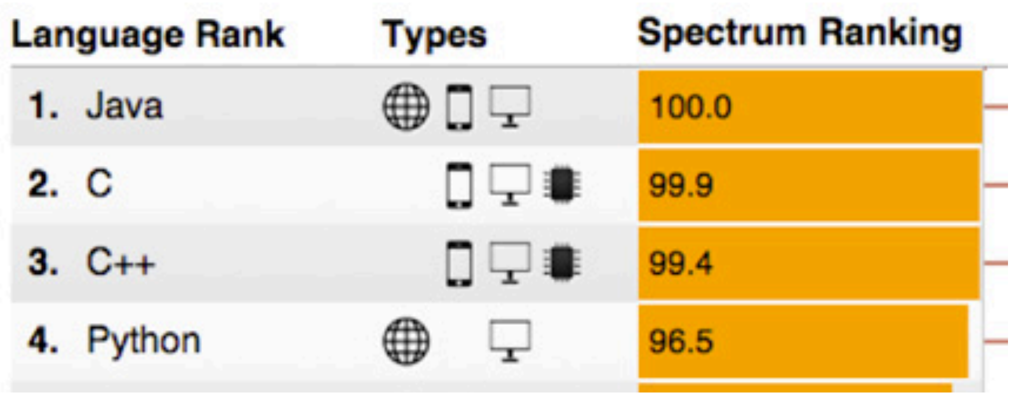
- Provides a different way of decomposing programs
- Basic principles:
  - Encapsulation of local, mutable state
  - Inheritance to share code
  - Dynamic dispatch to select which code gets run

- but why specifically Java?

Important Ecosystem

- Canonical example of OO language design
- Widely used: Desktop / Server / Android / etc.

- Industrial strength tools
  - Eclipse
  - JUnit testing framework
  - Profilers, debuggers, …



Language Rank	Types	Spectrum Ranking
1. Java		100.0
2. C		99.9
3. C++		99.4
4. Python		96.5

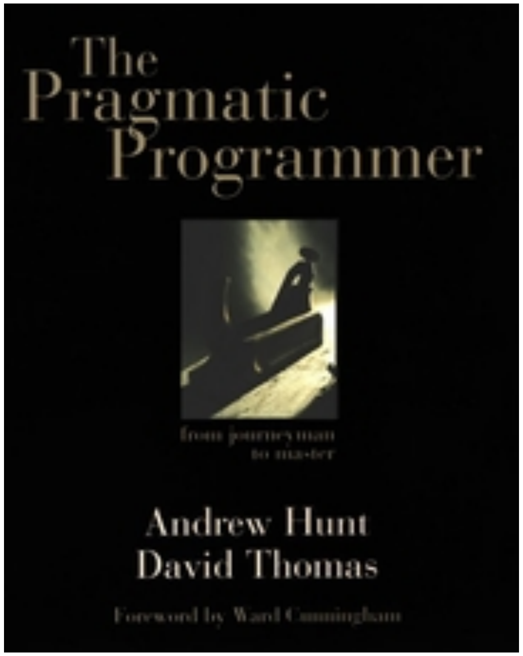
- Libraries:
  - Collections
  - I/O libraries
  - Swing
  - …

Onward…

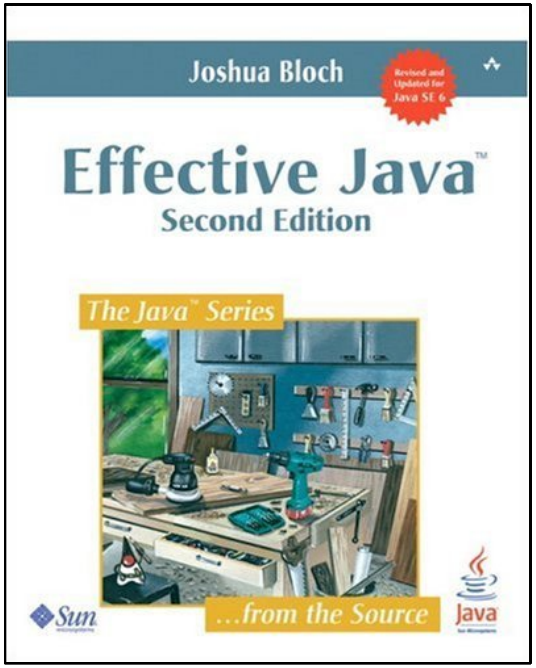
What Next?

- 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
  - CIS 552 – advanced programming
  - And many more!

The Craft of Programming



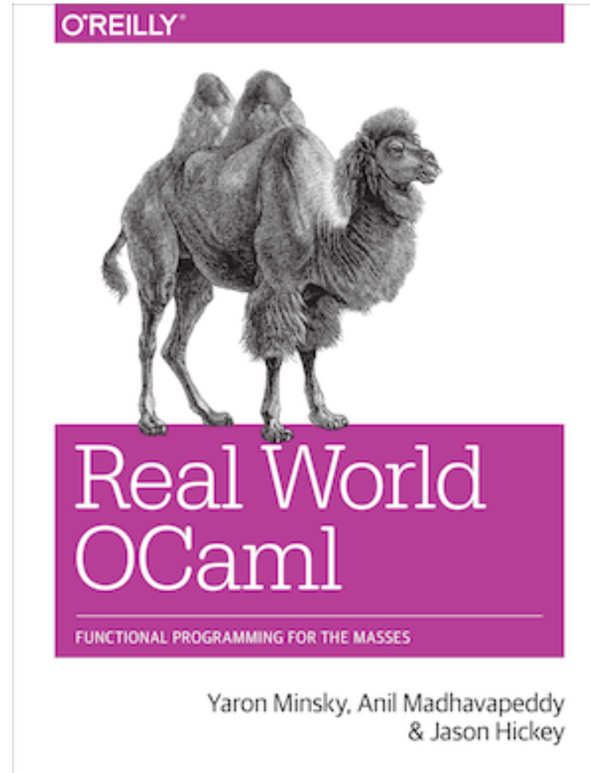
- *The Pragmatic Programmer: From Journeyman to Master*
  by Andrew Hunt and David Thomas
  - Not about a particular programming language, it covers style, effective use of tools, and good practices for developing programs.



- *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.

Craft of Programming

- *Real World OCaml*
  by Yaron Minsky, Anil Madhavpeddy,
  and Jason Hickey
  - Using OCaml in practice: learn how to leverage its rich types, module system, libraries, and tools to build reliable, efficient software.
  - https://realworldocaml.org/



- Explore related Languages:

Haskell Scala Clojure

Ways to get Involved

dining philosophers
UNIVERSITY OF PENNSYLVANIA | COMPUTER SCIENCE CLUB

PENN
APPS

WiCS
RESIDENTIAL PROGRAM

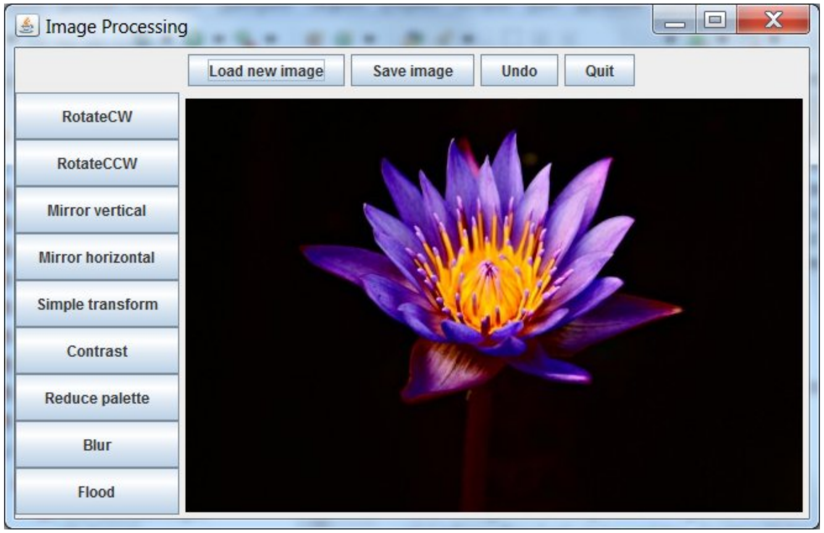
Women in Computer Science

Undergraduate Research

Become a TA!

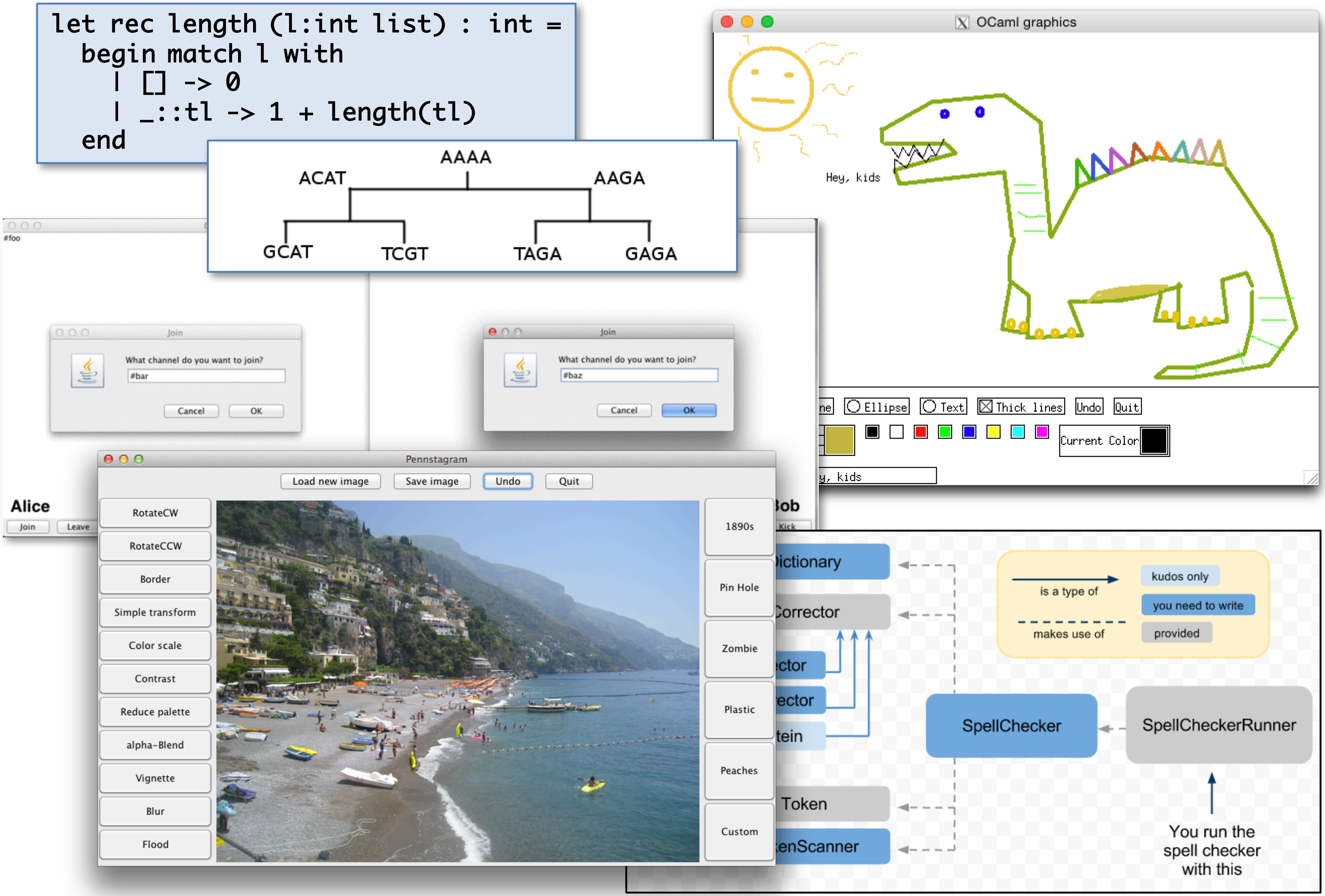
Parting Thoughts

- 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!



Thanks!

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



Did you attend class today?

1. yes
2. yes
3. yes
4. yes
5. maybe

The Billion Dollar Mistake

"I call it my billion-dollar mistake. It was the invention of the null reference in 1965. *At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement.* ***This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.*** *In recent years, a number of program analysers like PREfix and PREfast in Microsoft have been used to check references, and give warnings if there is a risk they may be non-null. More recent programming languages like Spec# have introduced* ***declarations for non-null references****. This is the solution, which I rejected in 1965. "*

Sir Tony Hoare, QCon, London 2009

Better interfaces: Optional values

- In Java, optional values are the default. *Any* reference type could be null.

- In OCaml, references are non-null by default and optional values must be specified by the programmer. Only values of type 'a option can be None.

- In Java, every method must specify what it does if its arguments are null. Many of them don't.
- In OCaml, the type of a method tells you whether an argument may be null. We didn't have to think about optional values until homework 5!

Fundamental abstract types

- An *abstract* type is defined by its *interface* not its *implementation*.
- Flexibility: interface can change without modification to clients
- Security: implementation invariants can be preserved

- In OCaml, direct expression of abstract data types through modules and signatures
- In Java, make types abstract via access modifiers (private), provide flexibility through interfaces