Programming Languages and Techniques (CIS120)

Lecture 39

Semester Recap

Announcements

- HW9: Game – Due *tonight* at 11:59pm

Final Exam:

- Tuesday, December 17th 6:00-8:00 PM
- 4 Locations:
  - Meyerson B1 — Last Names A - L
  - Towne 100 — Last Names M - Sh
  - Skirkanich Auditorium — Last Names Si – W
  - Moore 216 — Last Names X - Z

- Makeup exam offered Weds. December 18th
  - required by registrar for exam conflicts
  - time: noon – 2:00PM
  - sign up form on course web page

Exam Preparation

- *Comprehensive* exam over course concepts:
  - Ideas from OCaml material (no need to write OCaml)
  - All Java material (emphasizing material since midterm 2)
  - All course content *except lecture 36 (Code is Data)*

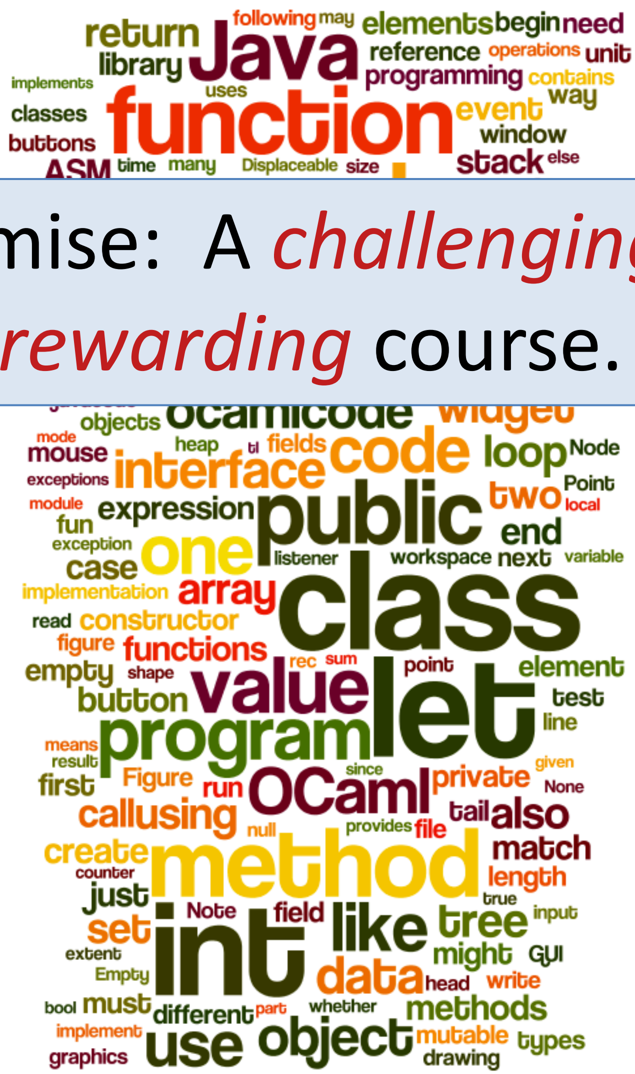
- Closed book, but:
  - You may use one letter-sized, two-sided, *handwritten* sheet of notes during the exam.

- Mock Exam and Review Session
  - Wednesday, December 11th 1:00-4:30PM
  - Towne 100

CIS 120 Recap

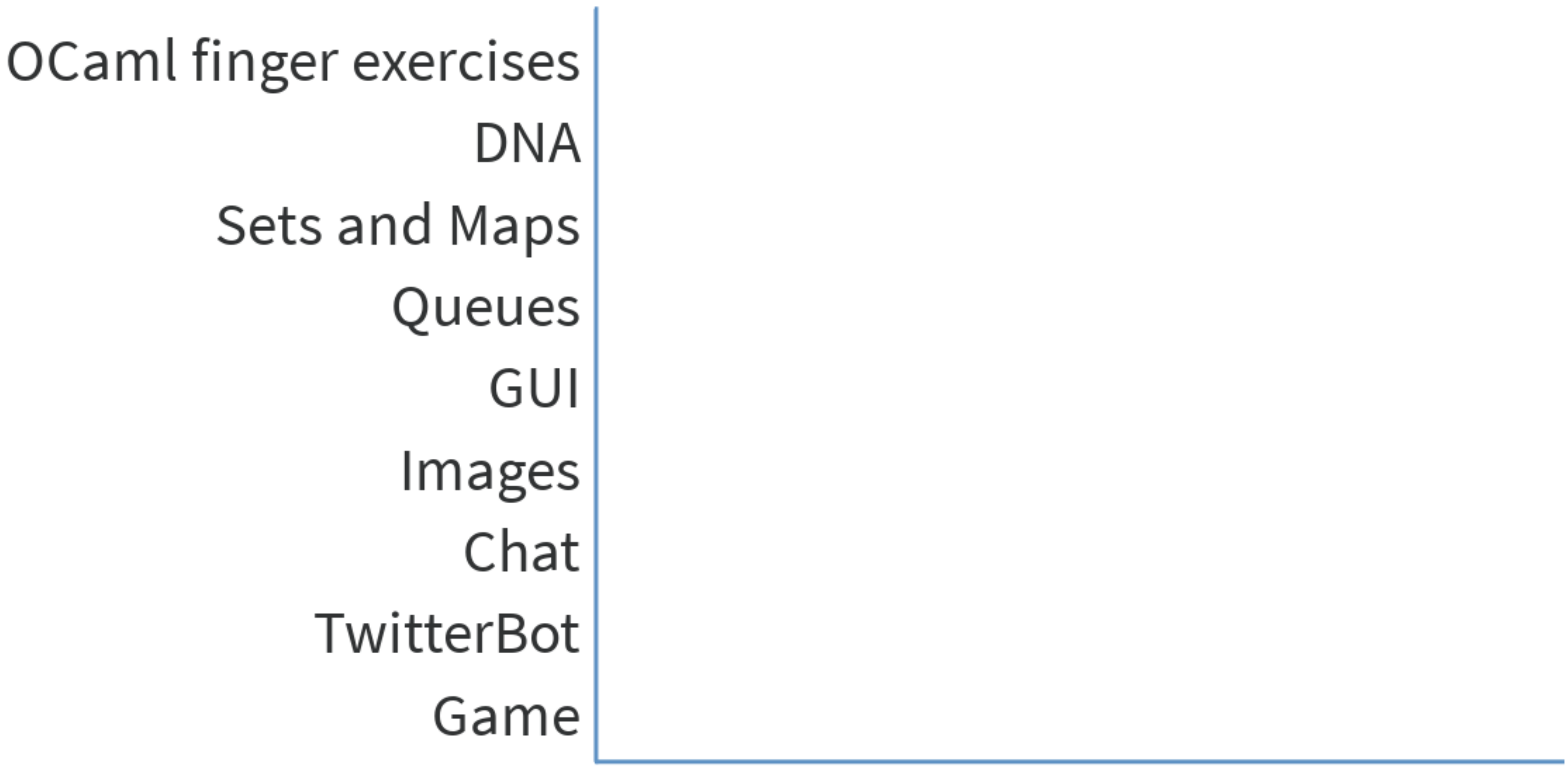
From Day 1

- 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!

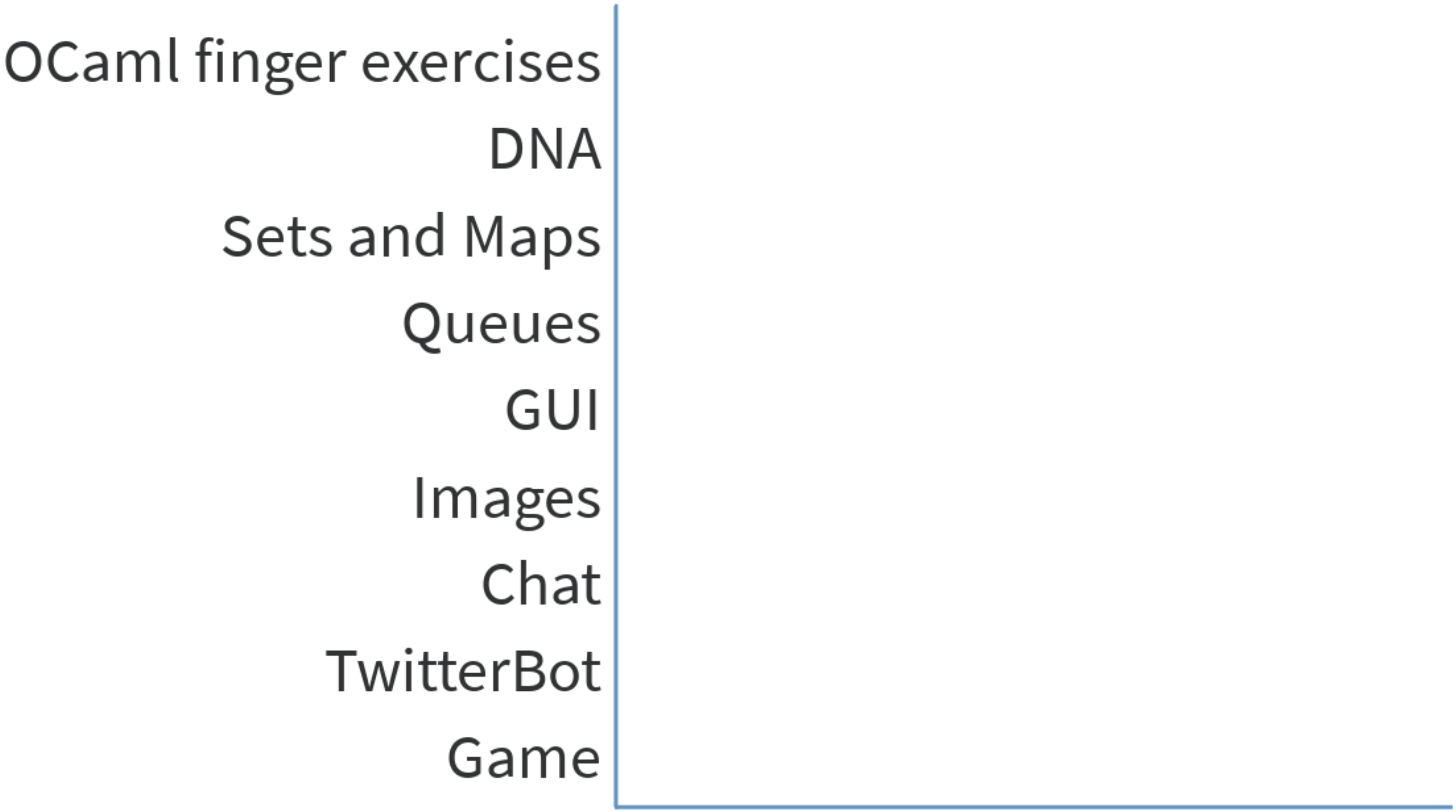


Promise: A *challenging* but *rewarding* course.

Which assignment was the most *challenging*?



Which assignment was the most *rewarding*?



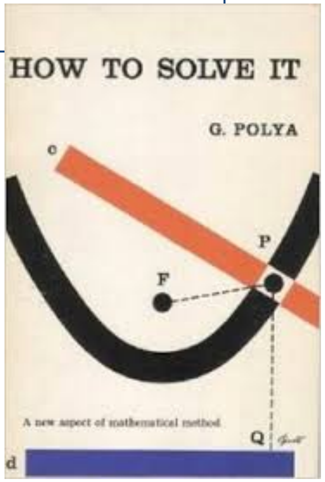
CIS 120 Concepts

13 concepts in 38 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."



Interface vs. Implementation

- 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

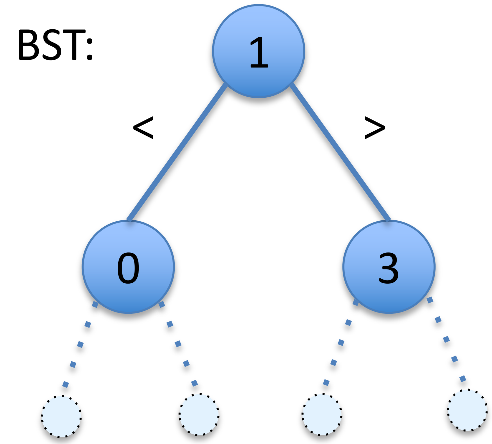
- Examples: Set/Map interface (HW3), queues in
  and access

entation without

about the

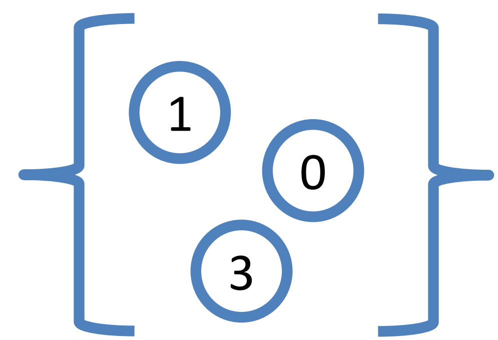
Invariants are a crucial tool for reasoning about data structures:

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



concrete representation

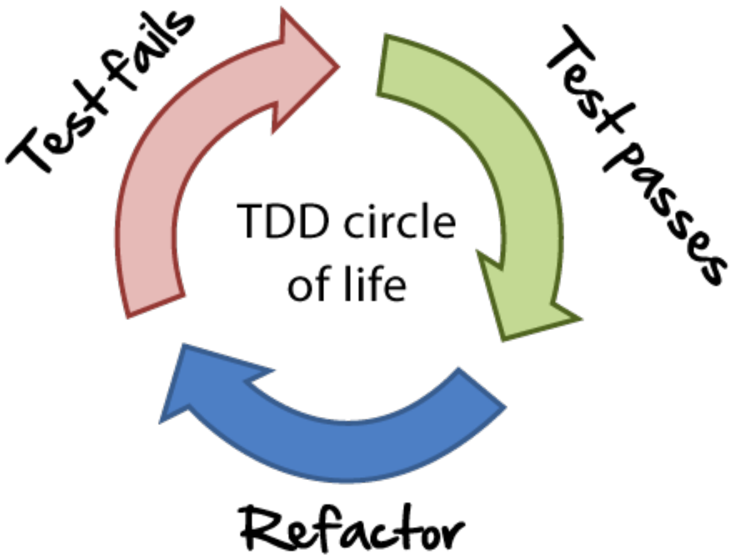
abstract view



Testing

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

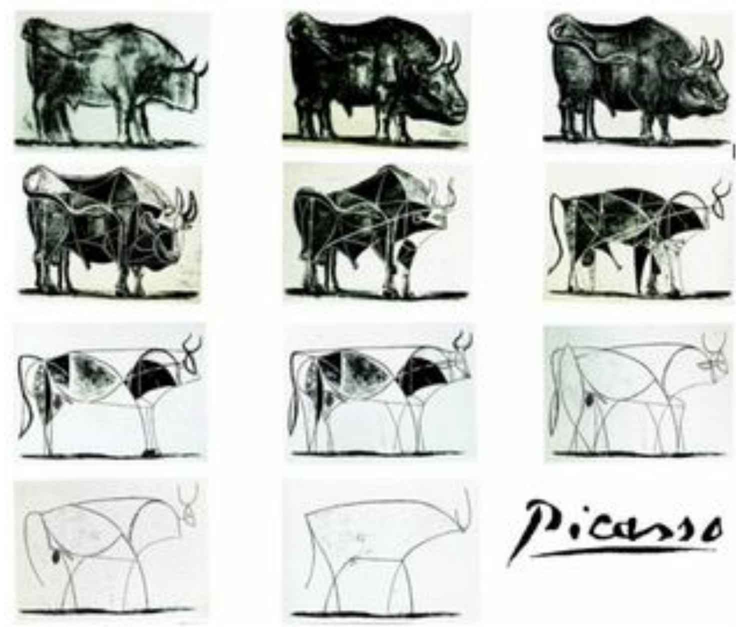
- Examples:
  - Simple assertions for declarative programs (or subprograms)
  - Longer (and more) tests for stateful programs / subprograms
  - Informal tests for GUIs (can be automated through tools)



- Why?
  - Tests clarify the specification of the problem
  - Helps you understand the *invariants*
  - Thinking about tests informs the implementation
  - Tests help with extending and refactoring code later
  - Industry practice; useful for coordinating teams

Functional/Procedural Abstraction

- Concept: *Don't Repeat Yourself!*
  - 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 *trar*
structures
- Examples: immutable lists anc
images, Strings, Streams in Jav
- Why?
  - Simple model of comp   , si
  - Simple interface: D   ave 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

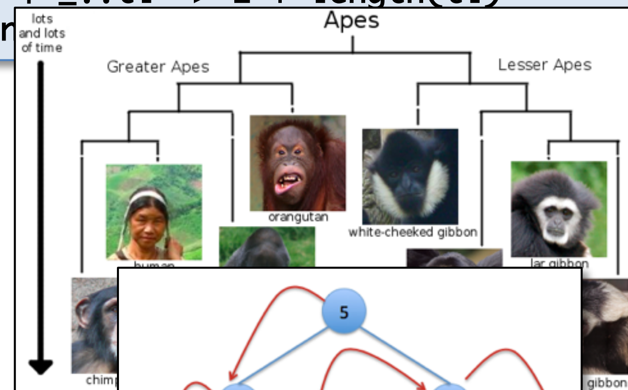
Recursion is the natural way of computing a function *f(t)* when *t* belongs to an inductive data type:

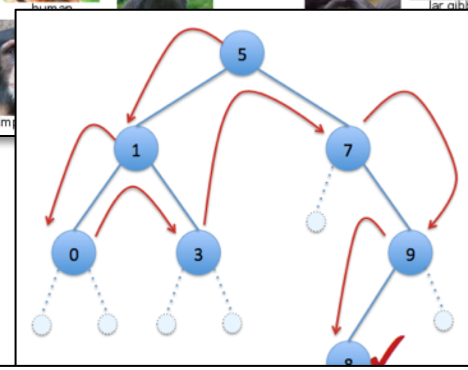
1. Determine the value of *f* for the base case(s).
2. Compute *f* for larger cases by combining the results of recursively calling *f* on smaller cases.
3. Same idea as mathematical induction (a la CIS 160)

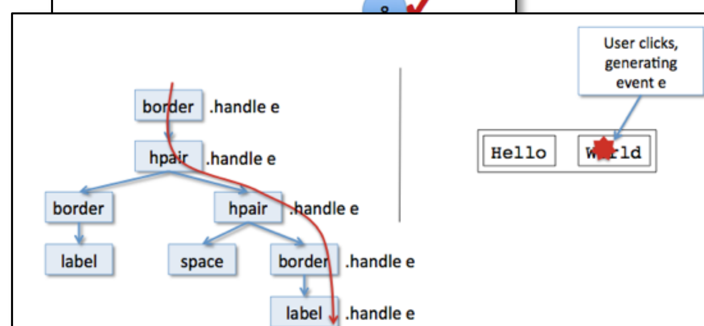
Concept: Tree Structured data

- 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)
  en
```







- 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 voi
        select(
    }
});
```

```
cell.addMouseListener(e -> {
            selectCell(cell);
});
```

- Why?
  – Powerful tool for abstraction: can factor out design patterns that differ only in certain computations

Types, Generics, and Subtyping

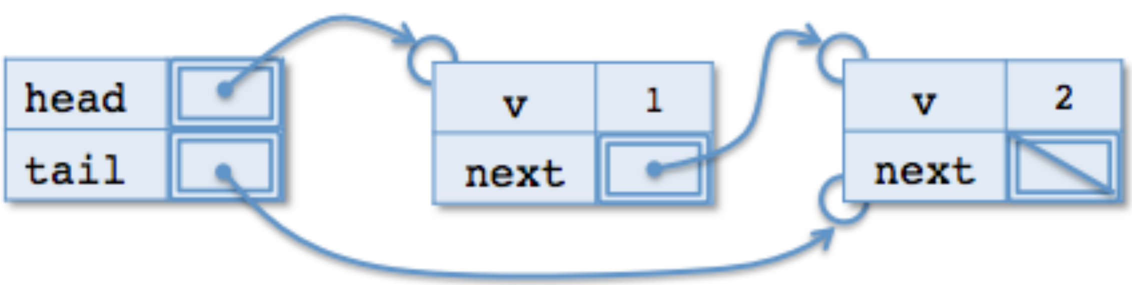
- Concept: *Static type systems* prevent many errors. Every expression has a static type, and OCaml/Java use the types to rule out buggy programs. *Generics* and *subtyping* make types more flexible and allow for better code reuse.

```
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

Mutable data

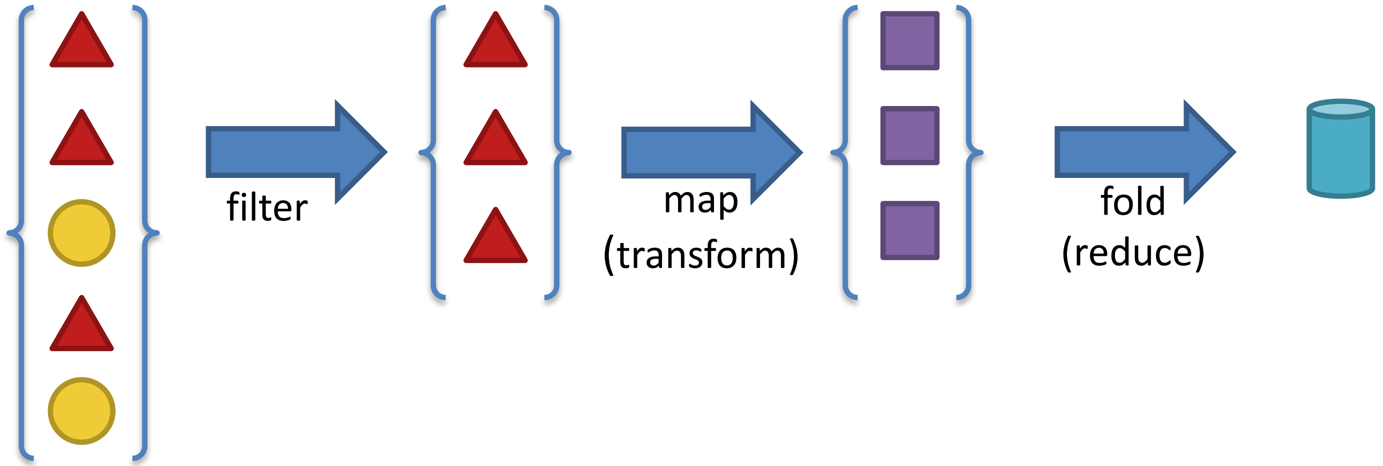
- Concept: Some data structures are *ephemeral*: computations mutate them over time
- Examples: queues, deques (HW4), GUI state (HW5, 9), arrays (HW 6), dictionaries (HW8)
- Why?
  - Common in OO programming, which simulates the transformations that objects undergo when interacting with their environment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

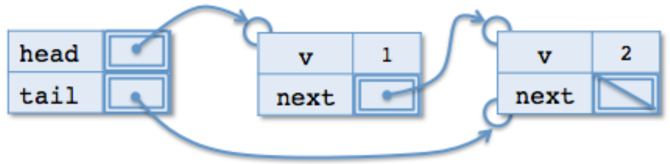
- Concept: Specific collection types: *sequences, sets*, and *finite maps*
- Examples: HW3, Java Collections, HW 7, 8
- Why?
  - These abstract data types come up again and again
  - Need *aggregate* data structures (collections) no matter what language you are programming in
  - Need to be able to choose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

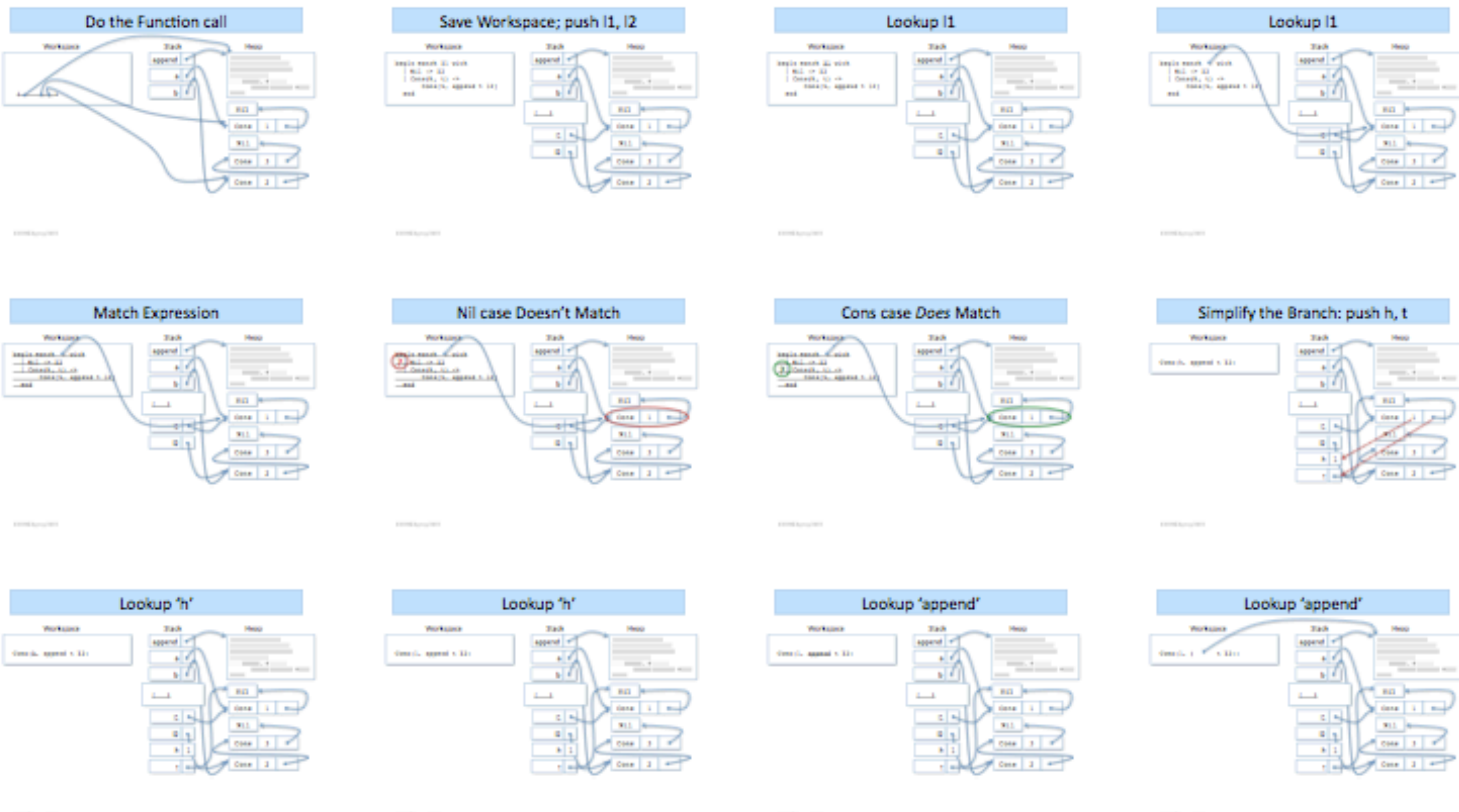
interface Set {boolean isEmpty(); …}



A queue with two elements

Abstract Stack Machine

- Concept: The *Abstract Stack Machine* is a detailed model of how programs execute in OCaml/Java



Abstract Stack Machine

- Concept: The *Abstract Stack Machine* is a detailed model of how programs execute in OCaml/Java

- Example: Many, throughout the semester!

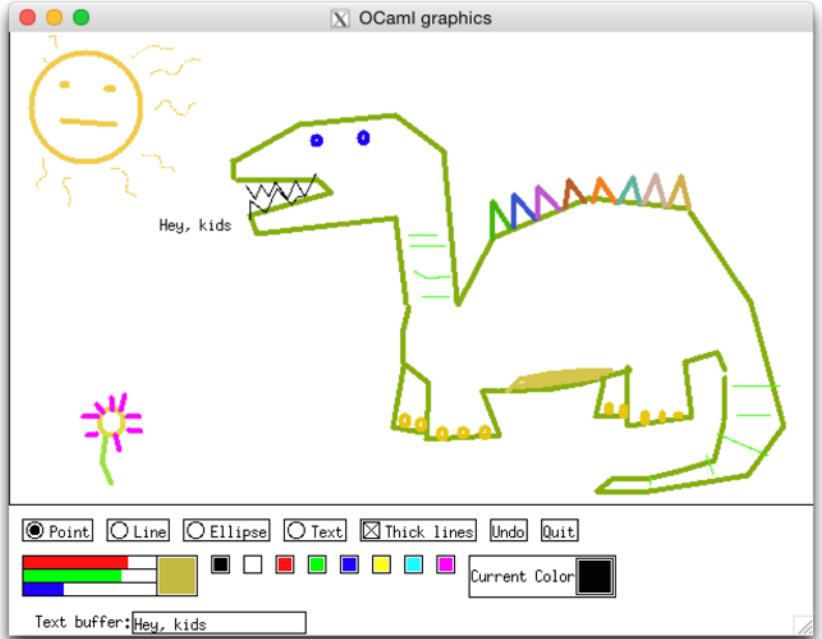
- Why?
  - To know what your program does without running it
  - To understand tricky features of Java/OCaml language (aliasing, first-class functions, exceptions, dynamic dispatch)
  - To help understand the programming models of other languages: Javascript, Python, C++, C#, …
  - To help predict performance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

Event-Driven programming

- Concept: Structure a program by associating "handlers" that *react to 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
  - Basis 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 — allows us to emphasize language-independent concepts
- Learn concepts that generalize *across* diverse languages.

…but why *specifically* OCaml?

OCaml

Rich, orthogonal vocabulary

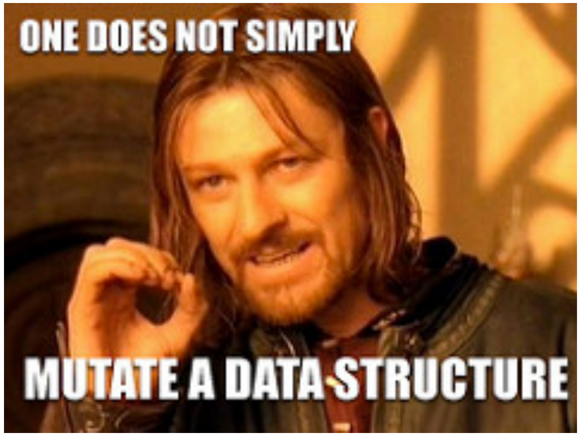
- In Java: int, A[], Object, Interfaces
- In OCaml:
  - primitives
  - arrays
  - objects
  - datatypes (including lists, trees, and options)
  - records
  - refs
  - first-class functions
  - abstract types

- 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 in OCaml, with 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)
  - Simple but powerful evaluation model (substitution + recursion).



Why Java?

Object Oriented Programming

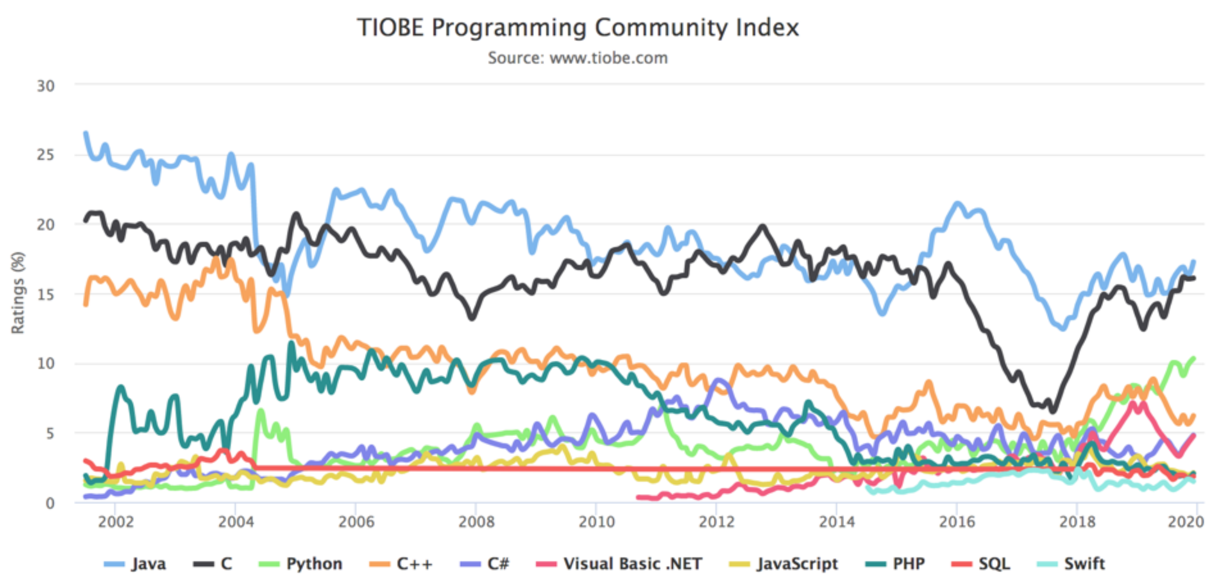
- A different way of decomposing / structuring programs
- Basic principles:
  - Encapsulation of local, mutable state
  - Inheritance to share code
  - Dynamic dispatch to select which code gets run
- but why *specifically* Java?

Important Ecosystem

- Canonical example of OO language design
- Widely used: Desktop / Server / Android / etc.
- Industrial strength tools
  - Eclipse
  - JUnit testing framework
  - Profilers, debuggers, …
- Libraries:
  - Collections / I/O
  - …
- In-demand job skill

IEEE Spectrum Rank

Rank	Language	Type	Score
1	Python		100.0
2	Java		96.3
3	C		94.4
4	C++		87.5
5	R		81.5
6	JavaScript		79.4



Onward…

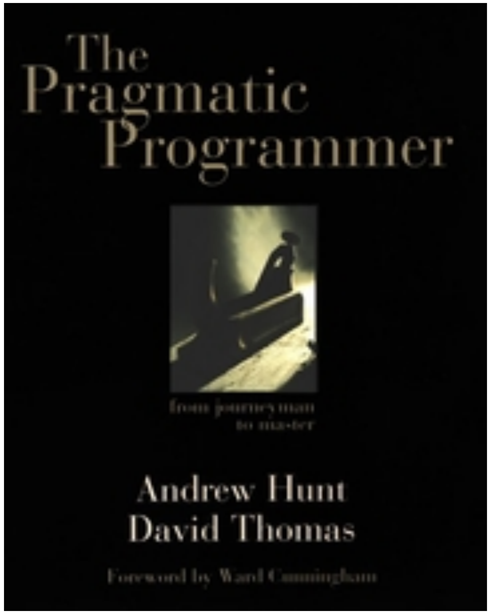
What Next?

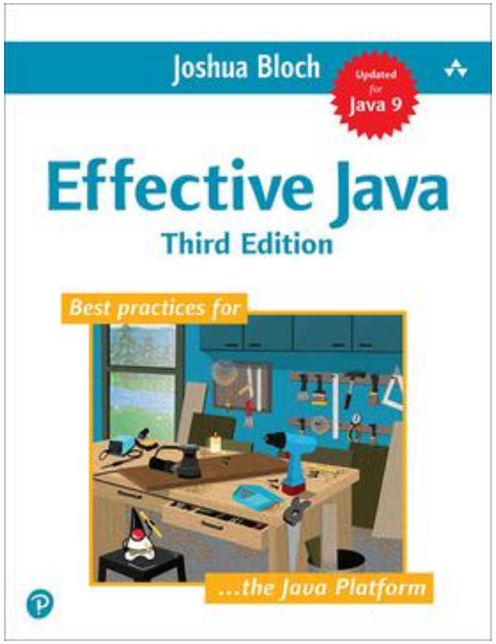
- Classes:
  - CIS 121, 262, 320 – data structures, performance, computational complexity
  - CIS 19x – programming languages
    - Python, Haskell, Ruby on Rails, iPhone programming, Android, Javascript, Rust
  - CIS 240 – lower-level: hardware, gates, assembly, C programming
  - CIS 341 – compilers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

Penn Engineering

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.

Functional 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 REASON

Kotlin Clojure F# Swift

Conferences / Videos / Blogs

- curry-on.org
- cufp.org Commercial Users of Functional Programming
  - See e.g. Manuel Chakravarty's talk "A Type is Worth a Thousand Tests"
- Yaron Minsky's Jane Street Tech Blog
  - Ocaml in practice
- PHASE – Philly Area Scala Enthusiasts
- Join us! Penn's PL Club plclub.org

Ways to get Involved

Computer Science Society
at the University of Pennsylvania

PENN APPS

PENN PL

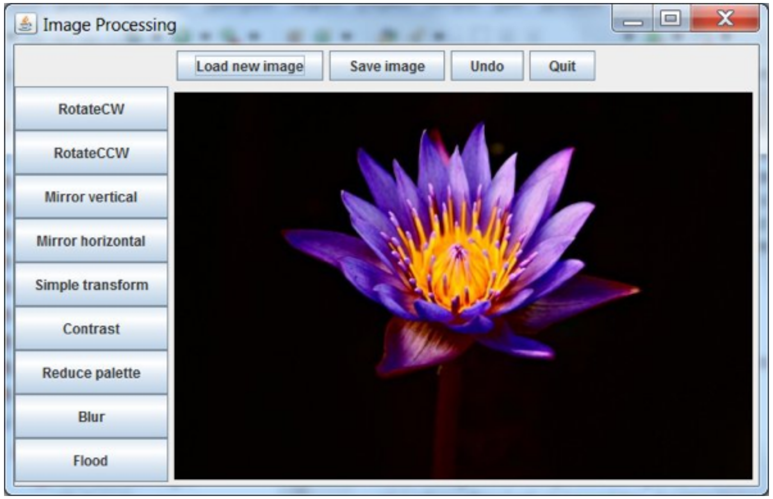
Become a TA!

</wics>
university of pennsylvania

Undergraduate Research

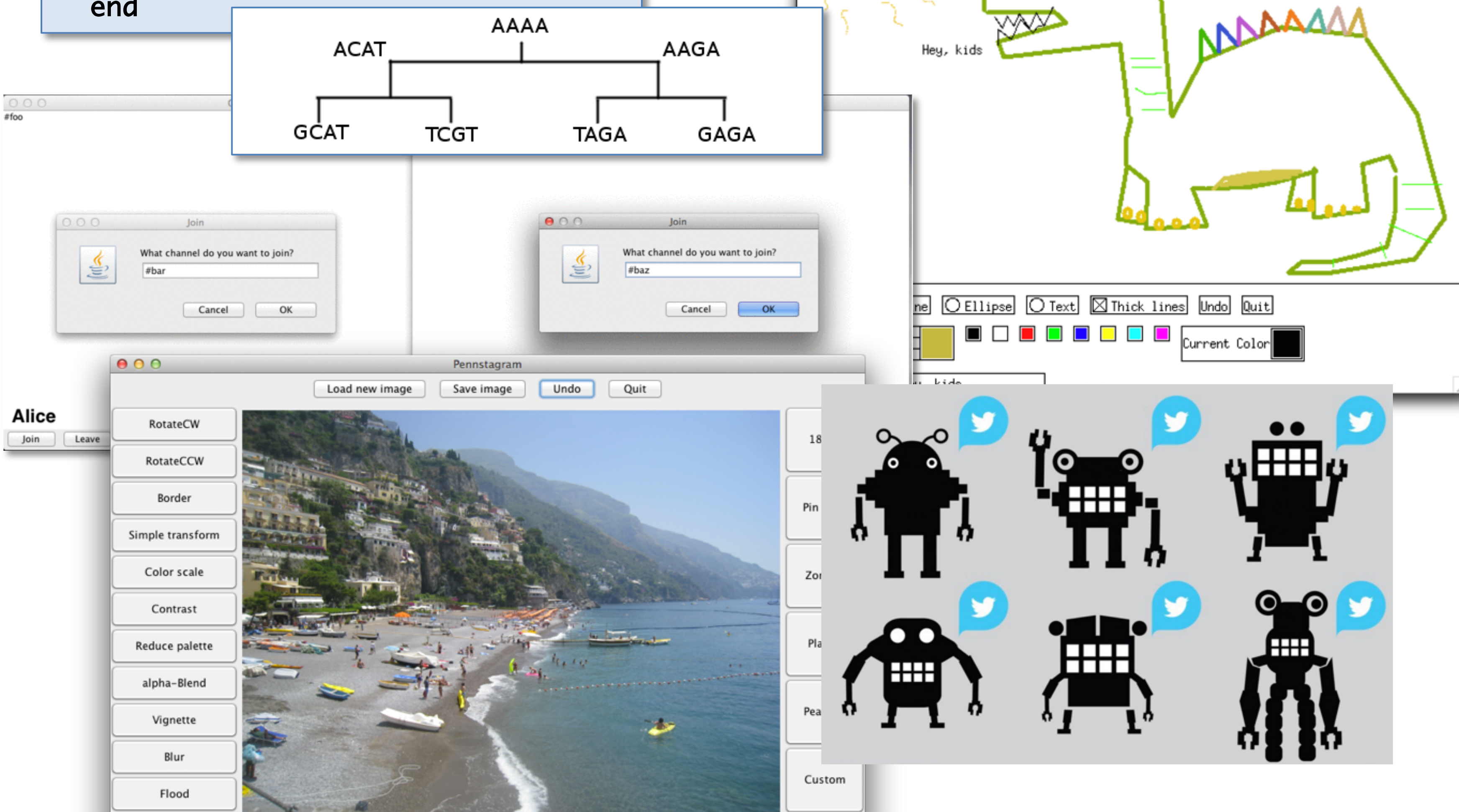
Parting Thoughts

- 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
```




CLUB RUGBY
CHAMPIONS