Programming Languages and Techniques (CIS1200)

Lecture 38

Semester Recap

Announcements

- Final Program Due: (92 points)

  Monday, December 9th at 11:59pm

  - Submit zipfile online, submission *only* checks if your code compiles
  - IntelliJ is **strongly recommended** for this project
  - You may distribute your game (after the deadline) if you do not use any of our code
- Grade based on demo with your TA during/after reading days
  - Grading rubric on the assignment website
  - Recommendation: don't be too ambitious.
- *NO LATE SUBMISSIONS*

CIS 1200 Final Exam

- Tuesday, December 17th 12:00-2:00 PM
  - Meyerson Hall B1 — Last Names A – M
  - Fagin Auditorium — Last Names N – Z

- Students who need accommodations should schedule their exams (ASAP) through the Weingarten Center

- Review Session / Mock exam
  - 2 hour mock exam followed by 2 hour review
  - (The review session will be recorded)
  - Location and Time – TBA
  - Look for details on Ed

Exam Preparation

- *Comprehensive* exam covering the entire course:
  - *Ideas* from OCaml material (but no need to write OCaml)
  - All Java material
    - emphasizing material since midterm 2: subtyping, dynamic dispatch, collections, equality & overriding, exceptions, I/O, inner classes, swing
  - All course content
    - *except:* Bonus Lecture *(Code is Data) and* Guest Lecture *(Jane Street)*
    - *Only simple/shallow questions about Wednesday's lecture*

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

Monday's Guest Lecture

Designing OCaml to be predictably faster at Jane Street

Speaker: Richard Eisenberg, Jane Street

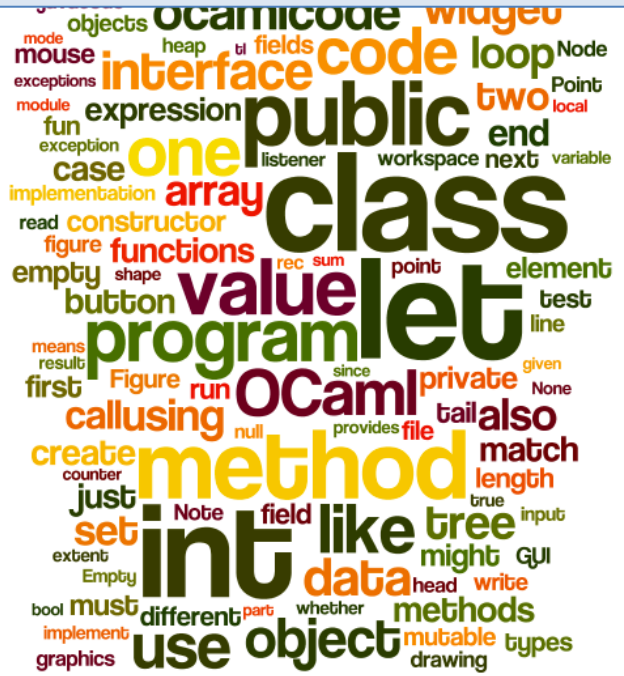
Abstract: Jane Street uses OCaml programs to power millions of market transactions daily. These programs must be fast, and also their performance must be consistent. The way polymorphism is designed in OCaml makes predictable high performance hard to achieve. This talk will explain why this is so, and how we plan to fix it by extending the language and compiler. Along the way, we'll see how careful study of programming languages yields tangible results in a practical setting. The talk will conclude with information about how OCaml is used at Jane Street and some of the exciting opportunities there.

CIS 1200 Recap

From Day 1

- CIS 1200 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*?

0

OCaml finger exercises 0%

DNA 0%

Sets and Maps 0%

Queues 0%

GUI 0%

Images 0%

Chat 0%

TwitterBot 0%

Game 0%

Which assignment was the most *rewarding*?

0

OCaml finger exercises 0%

DNA 0%

Sets and Maps 0%

Queues 0%

GUI 0%

Images 0%

Chat 0%

TwitterBot 0%

Game 0%

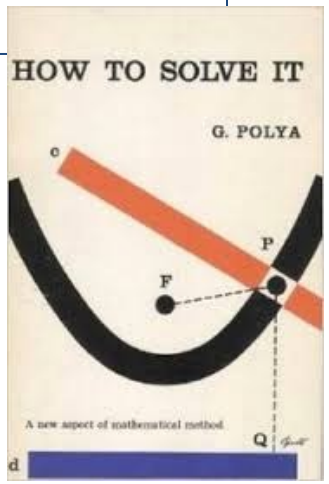
CIS 1200 Concepts

13 concepts in 36 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

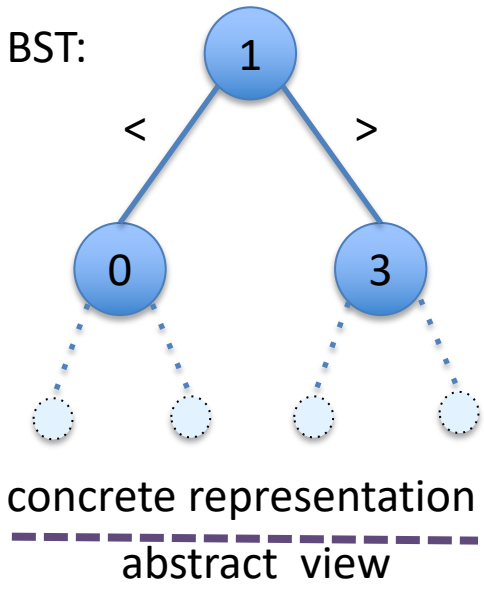
"Solving problems", wrote Polya, "is a practical art, like swimming, or skiing, or playing the piano: You can learn it only by imitation and practice."



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

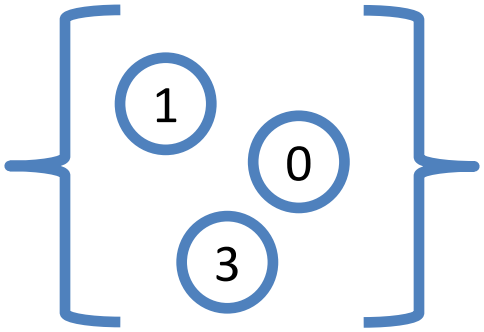


Invariants are a crucial tool for reasoning about data structures:

1. *Establish* the invariants when you create the structure.
2. *Preserve* the invariants when **you** modify the structure.
3. *Protect* the structure from external modification.

…entation without

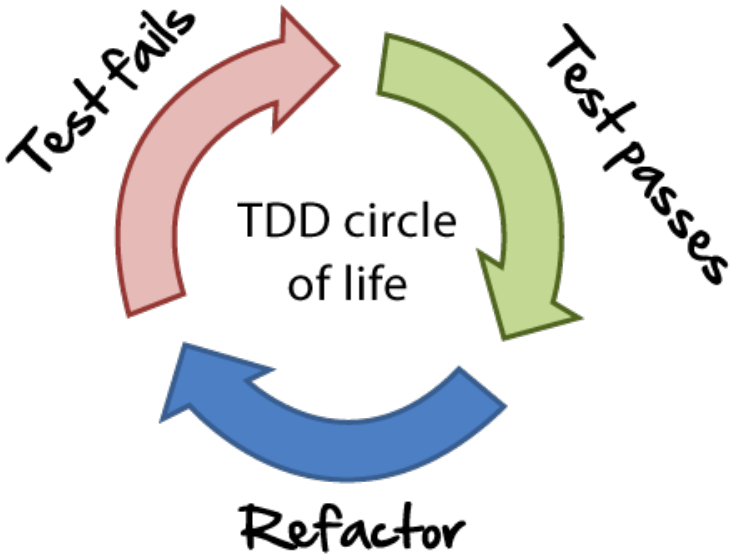
…about the



Testing

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

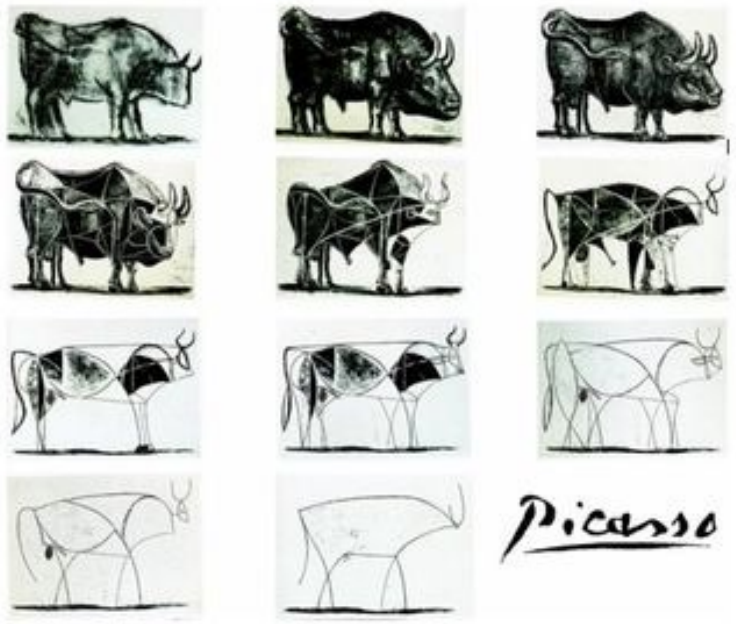
- Examples:
  - Simple assertions for declarative programs (or subprograms)
  - Longer (and more) tests for stateful programs / subprograms
  - Informal tests for GUIs (can be automated through tools)



- 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 inner 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 computations as *t*
- Examples: immutable lists ar
  images, Strings, Streams in Ja
- Why?
  - Simple model of co n
  - Simple interface t have to re
    communicati between various parts of the program, all interfaces are explicit)
  - *Recursion* amenable to mathematical analysis (CIS 1600/1210)
  - Plays well with concurrency

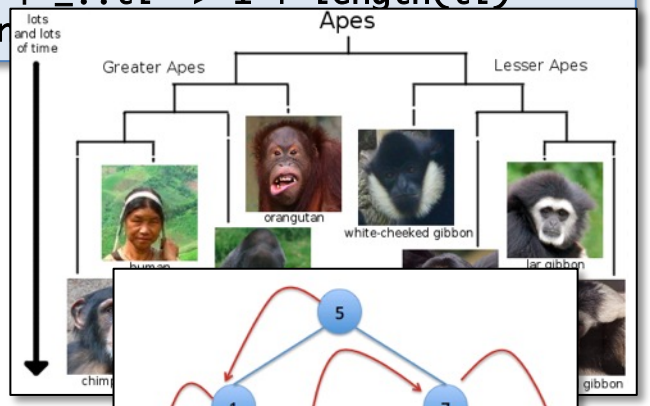
Recursion is the natural way of computing a function $f(t)$ when t belongs to an *inductive data* type:

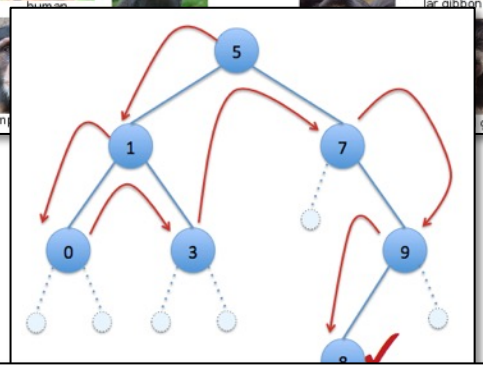
1. Determine the value of f for the base case(s).
2. Compute f for larger cases by combining the results of recursively calling f on smaller cases.
3. Same idea as mathematical induction (a la CIS 1600)

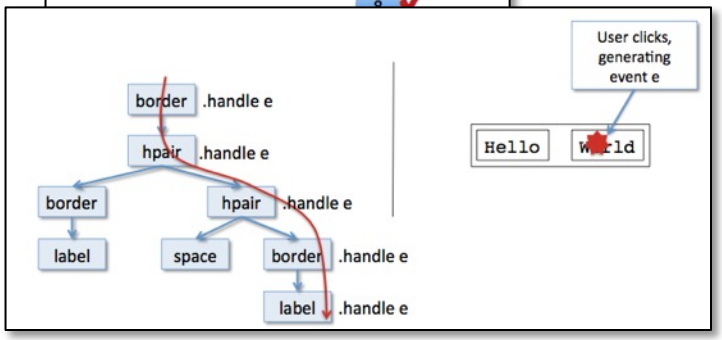
Concept: Tree Structured data

- Lists (i.e., “unary” trees)
- Simple binary trees
- Trees with invariants: e.g., binary search trees
- Widget trees: screen layout + event routing
- Swing components
- Trees are ubiquitous in computer science!
  - file system organization
  - languages, compilers
  - domain name hierarchy www.google.com
- Why?
  - organized data leads to divide and conquer algorithms that are more efficient

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







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
        selectC
    }
});
```

```
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* can detect many errors early. 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
```

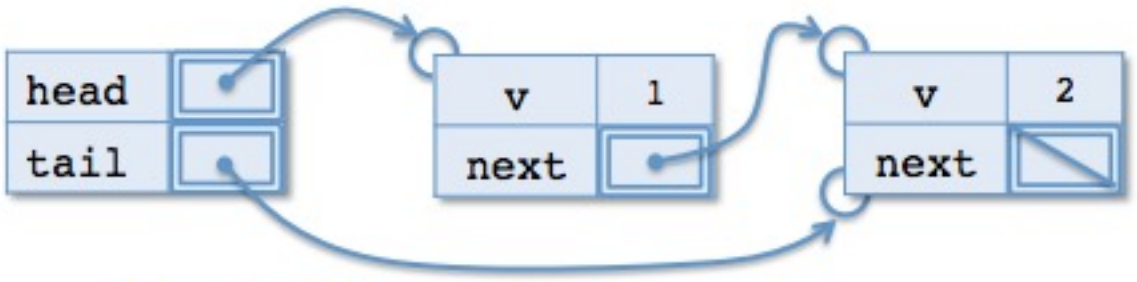
- Why?
  - Lets the language enforce (programmer-defined) abstraction
  - Easier to fix problems indicated by a type error than to write a test case and then figure out why the test case fails
  - Promotes refactoring: type checking ensures that basic invariants about the program are maintained

Mutable data

- Concept: Some data structures are *ephemeral*: computations mutate them over time

- Examples: queues, deques (HW4), GUI state (HW5, 9), arrays (HW 6), iterators (HW8)

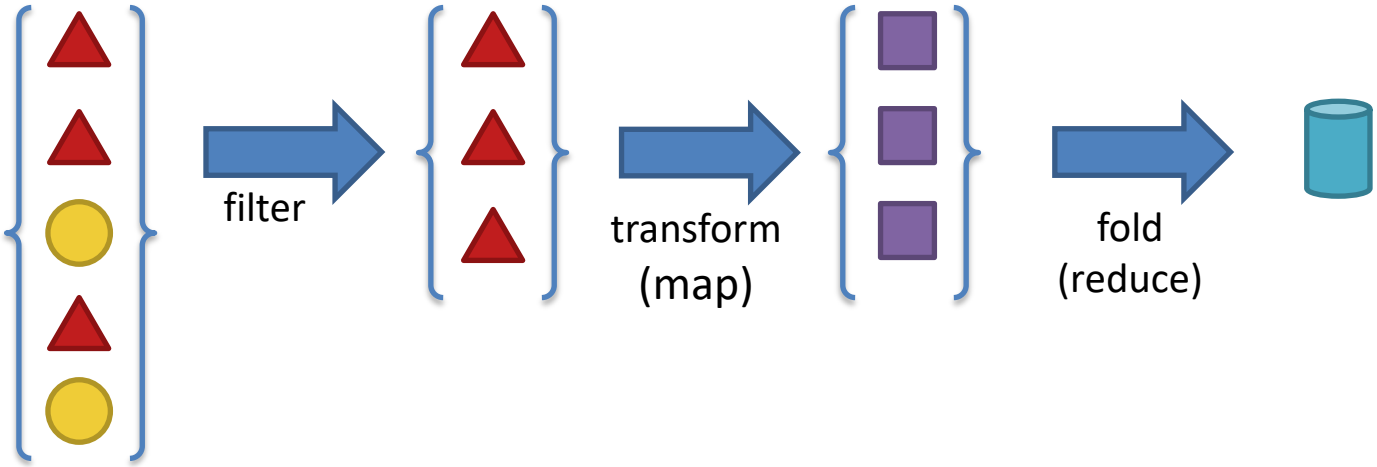
- Why?
  - Common in OO programming, which simulates the transformations that objects undergo when interacting with their environment
  - Heavily used for event-based programming, where different parts of the application communicate via shared state
  - Default style for Java libraries (collections,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



A queue with two elements

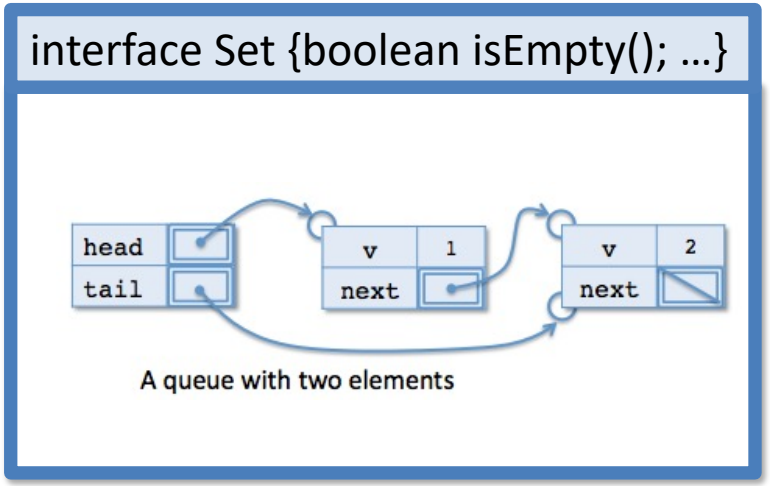
Abstract types: Sequences, Sets, Maps

- 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



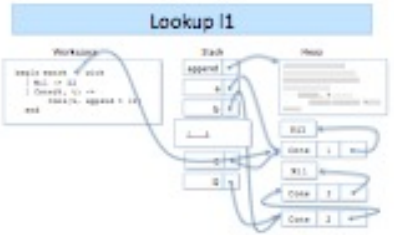
Lists, Trees, BSTs, Queues, and Arrays

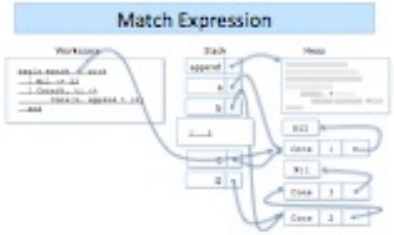
- Concept: There are *implementation trade-offs* for abstract types
- Examples:
  - Binary Search Trees vs. (linked) Lists vs. Hashing for sets and maps
  - Linked lists vs. Arrays for sequential data
- Why?
  - Abstract types have multiple implementations
  - Different implementations have different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

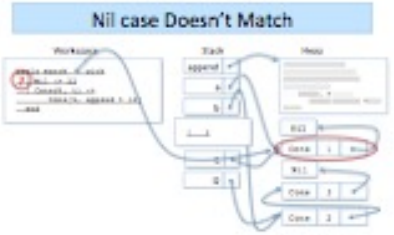


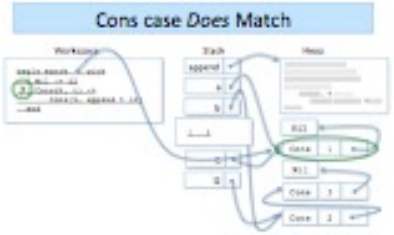
Abstract Stack Machine

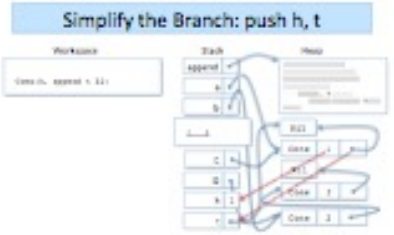
- Concept: The *Abstract Stack Machine* is a detailed model of how programs execute in OCaml/Java

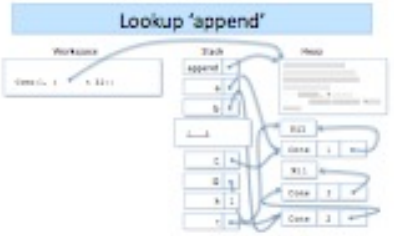












Abstract Stack Machine

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

- Example: Many, throughout the semester!

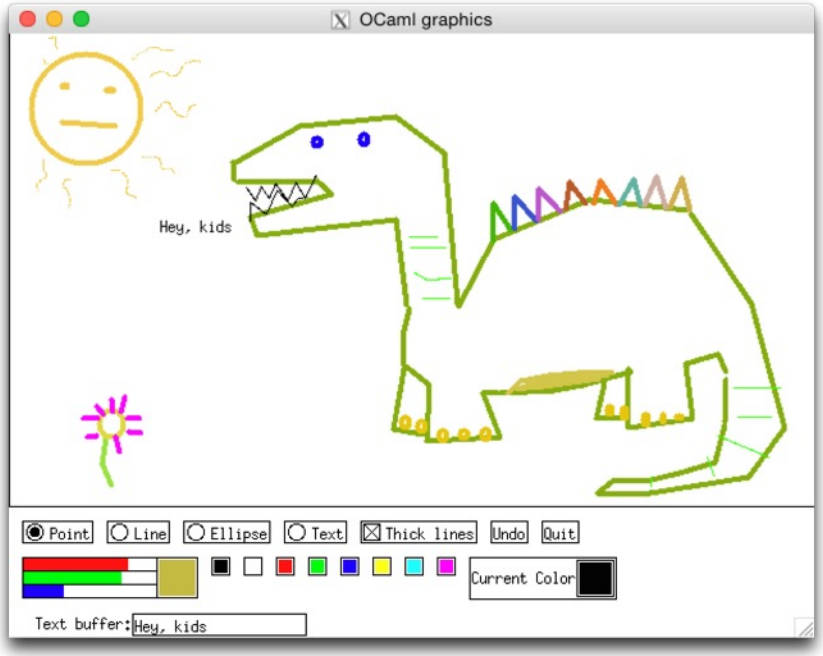
- Why?
  - To know what your program does without running it
  - To understand tricky features of Java/OCaml language (aliasing, first-class functions, exceptions, dynamic dispatch)
  - To help understand the programming models of other languages: Javascript, Python, C++, C#, …
  - To help predict performance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

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 — OCaml and Java — allows us to emphasize language-independent concepts
- Learn concepts that generalize *across* diverse languages.
- "OCaml-style" type systems have influenced many modern language designs

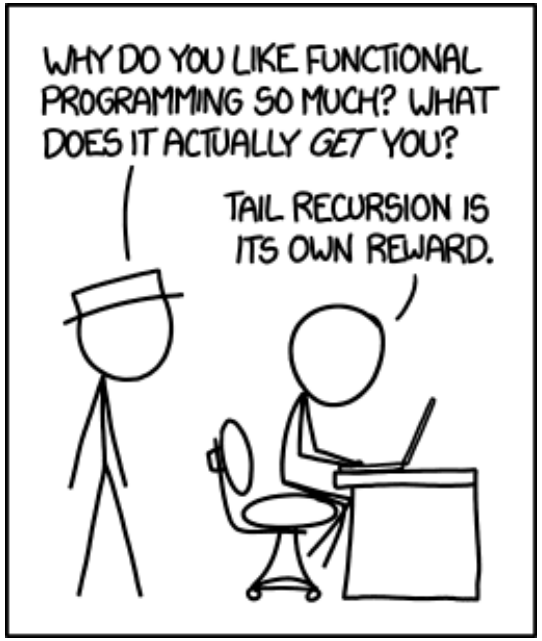
…but why *specifically* OCaml?

Rich, orthogonal vocabulary

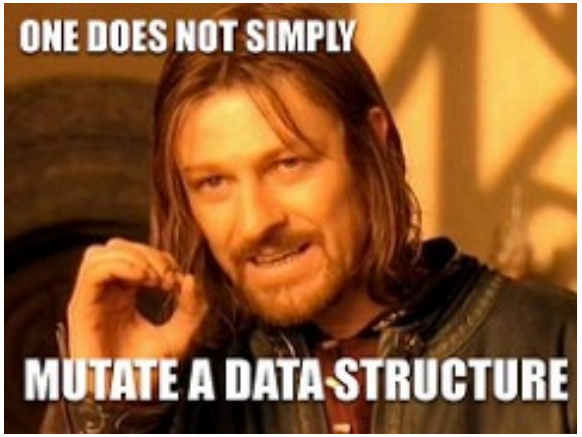
- In Java: int, A[], Object, Interfaces
- In OCaml:
  - primitives
  - arrays
  - objects
  - datatypes (including lists, trees, and options)
  - records
  - refs
  - first-class functions
  - abstract types
- All of the above *can* be implemented in Java, but untangling various use cases of objects is subtle
- Concepts like generics can be studied in isolation 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
  - Subtyping to capture statically known information about inheritance and the "*is a*" relationship

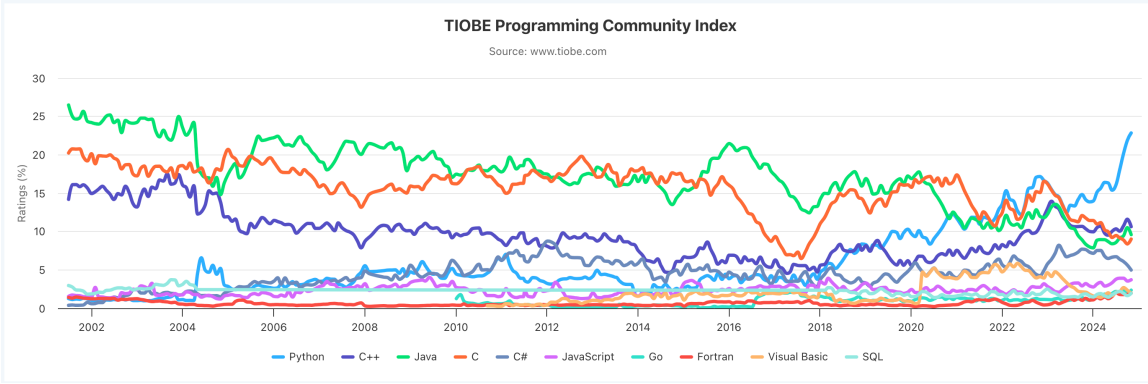
- but why *specifically* Java?

Important Ecosystem

- Canonical example of OO language design
- Widely used: Desktop / Server / Android / etc.
- Industrial strength tools
  - IntelliJ / Eclipse
  - JUnit testing framework
  - Profilers, debuggers, …
- Libraries:
  - Collections / I/O libraries/ Swing
  - …
- In-demand job skill
  - IEEE Spectrum: 2nd
  - TIOBE: 3rd

IEEE Spectrum Rank

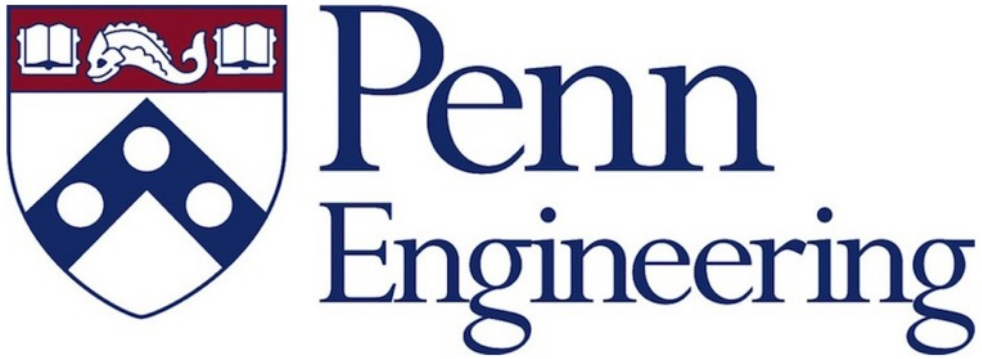
Rank	Language	Type	Score
1	Python		100.0
2	Java		96.3
3	C		94.4
4	C++		87.5
5	R		81.5
6	JavaScript		79.4
7	C#		74.5
8	Matlab		70.6



Onward…

What Next?

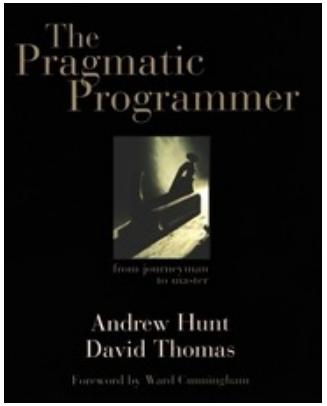
- Classes:
  - CIS 1210, 2620, 3200 – data structures, performance, computational complexity
  - CIS 19xx – programming languages
    - C++, Python, Haskell, Ruby on Rails, iPhone programming, Android, Javascript, Rust, Go
  - CIS 2400 – lower-level: hardware, gates, assembly, C programming
  - CIS 4710, 4480 – hardware and OS's
  - CIS 5520 – advanced functional programming in Haskell
  - CIS 5521 – compilers (projects in OCaml)
  - And many more!

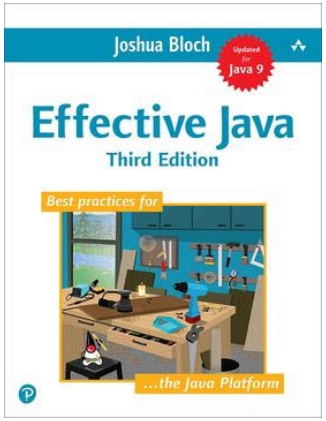


The Craft of Programming

- ***The Pragmatic Programmer:***
  From Journeyman to Master

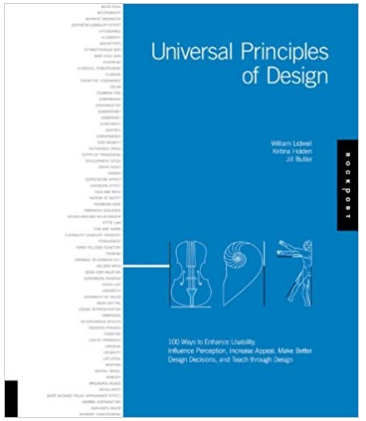
  by Andrew Hunt and David Thomas
  - Not about a particular programming language, it covers style, effective use of tools, and good practices for developing programs





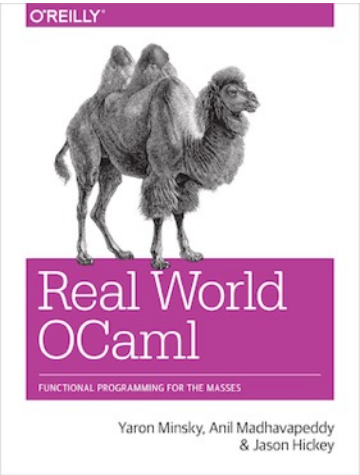
- ***Effective Java***
  by Joshua Bloch
  - Technical advice and wisdom about using Java for building software. The views we have espoused in this course share much of the same design philosophy

- **Universal Principles of Design**
  by William Lidwell, Kritina Holden, Jill Butler
  - General principles about good design with examples and applications ranging across software and user interfaces, to physical objects, to traditional graphic design.



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

- Many blogs / tutorials about Java
- curry-on.org
- cufp.org Commercial Users of Functional Programming
  - See e.g. Manuel Chakravarty's talk "A Type is Worth a Thousand Tests"
- Jane Street Tech Blog
  - OCaml in practice
  - "Building better software" podcast
- Join us! Penn's PL Club plclub.org

Ways to get Involved

Become a TA!

Undergraduate Research

Parting Thoughts

Help us improve CIS 1200!

- End-of-term survey will be sent soon
- Penn Course evaluations also provide useful feedback
- We take them seriously: please give us your thoughts!

Thanks to our amazing staff!

Thanks to you!

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