Functional Programming

Introduction To Cool
Cunning Plan

- ML Functional Programming
  - Fold
  - Sorting
- Cool Overview
  - Syntax
  - Objects
  - Methods
  - Types
CS 4501 - Compilers Practicum

- **Thursdays 2:00 to 3:00, Olsson 005**

- To be enrolled in CS 4501 (Compilers Practicum) you **must** be able to attend its listed lecture time.

- **First Meeting: This Thursday!**
  - Monday, January 23rd
PS1c Submission Statistics

- Students Taking Class for Credit: 58
- Students Submitting > 0 Times: 53
  - Note “Testing” vs. “Grading” submission
- Language choice, as Tuesday morning
  - Python 50
  - Ruby 47
  - JavaScript 35
  - C 18
  - OCaml 16
  - Cool 4

Which of these languages is the most important in the course?
PS1 Pedagogy

- Why target old languages?
  - Python 2.4 vs. 2.6, Ruby 1.8.5 vs. 1.9, etc.

- Real-world customer machine scenario

- Exposure to costs of adding language features
  - \{C, Ocaml, Cool\} vs. \{Python, Ruby\}, specs

- "Toposort Algorithm" vs. "Language, Syntax, Run-Time System, Operating System, Testing and Debugging"
  - "Whitespace doesn't matter" vs. "You write printf"

- Black box testing and debugging
  - http://www.st.cs.uni-saarland.de/dd/
  - http://www.whyprogramsfail.com/
# RUBY: Reverse-sort the lines from standard input
lines = [] # a list variable to hold all the lines we'll read in
working = true # are there still more lines to read in?
while working
    line = gets # read a line from standard input
    if line == nil # nil is "nothing, it didn't work"
        working = false # we're done reading stuff
    else
        lines[lines.length] = line # append 'line' to the end of 'lines'
    end # end of 'if'
end # end of 'while'
sorted = lines.sort do |a,b|
    # sort the list of lines
    # this do block is basically an anonymous function!
    # |foo,bar| means "foo and bar are the arguments"
    # we will tell it how to compare to arbitrary elements, a and b
    b <=> a # <=> means "compare" -- we'll do it in reverse
end # end 'do'
sorted.map{|one_line, i| # iterate over each statement in sorted list
    puts one_line # write it to standard output
} # end 'iteration'
This is my final day

• ... as your ... companion ... through Ocaml and Cool. After this we start the interpreter project.

• Clearly a third day would just be unthinkable.
One-Slide Summary

• Functions and type inference are **polymorphic** and operate on more than one type (e.g., List.length works on int lists and string lists).

• **fold** is a powerful higher-order function (like a swiss-army knife or duct tape).

• **Cool** is a Java-like language with classes, methods, private fields, and inheritance.
Pattern Matching (Error?)

- Simplifies Code (eliminates ifs, accessors)

```plaintext
type btree = (* binary tree of strings *)
  | Node of btree * string * btree
  | Leaf of string

let rec height tree = match tree with
  | Leaf _ -> 1
  | Node(x,_,y) -> 1 + max (height x) (height y)

let rec mem tree elt = match tree with
  | Leaf str | Node(_,str,_) -> str = elt
  | Node(x,_,y) -> mem x elt || mem y elt
```
Pattern Matching (Error?)

- Simplifies Code (eliminates ifs, accessors)

```ocaml
type btree = (* binary tree of strings *)
  | Node of btree * string * btree
  | Leaf of string

let rec height tree = match tree with
  | Leaf _ -> 1
  | Node(_,_,_) -> 1 + max (height x) (height y)

let rec mem tree elt = match tree with
  | Leaf str | Node(_,str,_) -> str = elt
  | Node(x,_,y) -> mem x elt || mem y elt
```

bug?
Pattern Matching (Error!)

• Simplifies Code (eliminates ifs, accessors)

type btree = (* binary tree of strings *)
  | Node of btree * string * btree
  | Leaf of string

let rec bad tree elt = match tree with
  | Leaf str | Node(_,str,_) -> str = elt
  | Node(x,_,y) -> bad x elt | bad y elt

let rec mem tree elt = match tree with
  | Leaf str | Node(_,str,_) when str = elt -> true
  | Node(x,_,y) -> mem x elt | mem y elt
Recall: Polymorphism

- Functions and type inference are **polymorphic**
  - Operate on more than one type
  - let rec length x = match x with
    - | [] -> 0
    - | hd :: tl -> 1 + length tl
  - val length : \(\alpha\) list -> int
  - length [1;2;3] = 3
  - length [“algol”; ”smalltalk”; ”ml”] = 3
  - length [1 ; “algol” ] = type error!

\(\alpha\) means “any one type”
Recall: Higher-Order Functions

- Function are first-class values
  - Can be used whenever a value is expected
  - Notably, can be passed around
  - Closure captures the environment
- let rec map f lst = match lst with
  - | [] -> []
  - | hd :: tl -> f hd :: map f tl
- val map : (α -> β) -> α list -> β list
- let offset = 10 in
- let myfun x = x + offset in
- val myfun : int -> int
- map myfun [1;8;22] = [11;18;32]

- Extremely powerful programming technique
  - General iterators
  - Implement abstraction

f is itself a function!
Recall: Fold

- The **fold** operator comes from Recursion Theory (Kleene, 1952)
  ```ocaml
define fold = 
  let rec fold f acc lst = match lst with
  | [] -> acc
  | hd :: tl -> fold f (f acc hd) tl
  in fold

- val fold : \(\alpha \to \beta \to \alpha\) -> \(\alpha\) -> \(\beta\) list -> \(\alpha\)

- Imagine we’re summing a list \((f =\text{addition})\):

  \[
  \begin{array}{c}
  9 \rightarrow 2 \rightarrow 7 \rightarrow 4 \rightarrow 5 \rightarrow \ldots \\
  \end{array}
  \begin{array}{c}
  \text{acc} \\
  \begin{array}{c}
  11 \rightarrow 7 \rightarrow 4 \rightarrow 5 \rightarrow \ldots \\
  \end{array}
  \end{array}
  \begin{array}{c}
  \text{lst} \\
  \begin{array}{c}
  18 \rightarrow 4 \rightarrow 5 \rightarrow \ldots \\
  \end{array}
  \end{array}
  \begin{array}{c}
  f \\
  \end{array}
  \begin{array}{c}
  27 \\
  \end{array}
  \]
Referential Transparency

- To find the meaning of a functional program we replace each reference to a variable with its definition.
  - This is called referential transparency.

- Example:

```plaintext
let y = 55
let f x = x + y
f 3
```

  --> means --> 3 + y

  --> means --> 3 + 55
Worked Example: Product

let rec fold f acc lst = match lst with
| [] -> acc
| hd :: tl -> fold f (f acc hd) tl

fold (*) 1 [8;6;7]
Worked Example: Product

```
let rec fold f acc lst = match lst with
| [] -> acc
| hd :: tl -> fold f (f acc hd) tl

fold (*) 1 [8;6;7]
```

match lst with
| [] -> acc
| hd :: tl -> fold f (f acc hd) tl
Worked Example: Product

```ocaml
let rec fold f acc lst = match lst with
| [] -> acc
| hd :: tl -> fold f (f acc hd) tl
```

```
fold (*) 1 [8;6;7]
```

with f=*, acc=1, and lst=[8;6;7]

```
match lst with
| [] -> acc
| hd :: tl -> fold f (f acc hd) tl
```
Worked Example: Product

let rec fold f acc lst = match lst with
| [] -> acc
| hd :: tl -> fold f (f acc hd) tl

fold (*) 1 [8;6;7]

match [8;6;7] with
| [] -> 1
| hd :: tl -> fold (*) (* 1 hd) tl
Worked Example: Product

let rec fold f acc lst = match lst with
| [] -> acc
| hd :: tl -> fold f (f acc hd) tl

match [8;6;7] with
| [] -> 1
| hd :: tl -> fold (*) (* 1 hd) tl
Worked Example: Product

```
let rec fold f acc lst = match lst with
  | [] -> acc
  | hd :: tl -> fold f (f acc hd) tl

let hd :: tl = [8;6;7] in
fold (*) (* 1 hd) tl
```

```
match [8;6;7] with
  | [] -> 1
  | hd :: tl -> fold (*) (* 1 hd) tl
```
Worked Example: Product

let rec fold f acc lst = match lst with
| [] -> acc
| hd :: tl -> fold f (f acc hd) tl

let hd :: tl = [8;6;7] in
fold (*) (* 1 hd) tl
Worked Example: Product

let rec fold f acc lst = match lst with
| [] -> acc
| hd :: tl -> fold f (f acc hd) tl

let hd :: tl = [8;6;7] in
fold (*) (* 1 hd) tl

fold (*) (* 1 8) [6;7]
let rec fold f acc lst = match lst with
| [] -> acc
| hd :: tl -> fold f (f acc hd) tl

fold (*) 8 [6;7]
Worked Example: Product

let rec fold f acc lst = match lst with
| [] -> acc
| hd :: tl -> fold f (f acc hd) tl

fold (*) 8 [6;7]

match lst with
| [] -> acc
| hd :: tl -> fold f (f acc hd) tl
Worked Example: Product

```ocaml
let rec fold f acc lst = match lst with
| [] -> acc
| hd :: tl -> fold f (f acc hd) tl
```

fold (*) 8 [6;7]

match [6;7] with
| [] -> 8
| hd :: tl -> fold (*) (* 8 hd) tl
Worked Example: Product

let rec fold f acc lst = match lst with
  | [] -> acc
  | hd :: tl -> fold f (f acc hd) tl

match [6;7] with
  | [] -> 8
  | hd :: tl -> fold (*) (* 8 hd) tl
Worked Example: Product

let rec fold f acc lst = match lst with
| [] -> acc
| hd :: tl -> fold f (f acc hd) tl

let hd :: tl = [6;7] in
fold (*) (* 8 hd) tl

match [6;7] with
| [] -> 8
| hd :: tl -> fold (*) (* 8 hd) tl
Worked Example: Product

let rec fold f acc lst = match lst with
  | [] -> acc
  | hd :: tl -> fold f (f acc hd) tl

let hd :: tl = [6;7] in
fold (*) (* 8 hd) tl
Worked Example: Product

let rec fold f acc lst = match lst with
| [] -> acc
| hd :: tl -> fold f (f acc hd) tl

let hd :: tl = [6;7] in
fold (*) (* 8 hd) tl

fold (*) (* 8 6) [7]
Worked Example: Product

```ocaml
let rec fold f acc lst = match lst with
| [] -> acc
| hd :: tl -> fold f (f acc hd) tl
```

fold (*) 48 [7]
Worked Example: Product

```ocaml
let rec fold f acc lst = match lst with
  | [] -> acc
  | hd :: tl -> fold f (f acc hd) tl

fold (*) 48 [7]
```

with $f=\times$, $acc=48$, and $lst=[7]$
let rec fold f acc lst = match lst with
| [] -> acc
| hd :: tl -> fold f (f acc hd) tl

fold (*) 48 [7]

match [7] with
| [] -> 48
| hd :: tl -> fold (*) (* 48 hd) tl
Worked Example: Product

```
let rec fold f acc lst = match lst with
| [] -> acc
| hd :: tl -> fold f (f acc hd) tl
```

```
match [7] with
| [] -> 48
| hd :: tl -> fold (*) (* 48 hd) tl
```
Worked Example: Product

let rec fold f acc lst = match lst with
| [] -> acc
| hd :: tl -> fold f (f acc hd) tl

let hd :: tl = [7] in
fold (*) (* 48 hd) tl

match [7] with
| [] -> 48
| hd :: tl -> fold (*) (* 48 hd) tl
let rec fold f acc lst = match lst with

| [] -> acc
| hd :: tl -> fold f (f acc hd) tl

let hd :: tl = [7] in
fold (*) (* 48 hd) tl
Worked Example: Product

let rec fold f acc lst = match lst with
| [] -> acc
| hd :: tl -> fold f (f acc hd) tl

fold (*) (* 48 7) []
Worked Example: Product

let rec fold f acc lst = match lst with
  | [] -> acc
  | hd :: tl -> fold f (f acc hd) tl

fold (*) 336 []

match lst with
  | [] -> acc
  | hd :: tl -> fold f (f acc hd) tl
Worked Example: Product

```ocaml
let rec fold f acc lst = match lst with
  | [] -> acc
  | hd :: tl -> fold f (f acc hd) tl
```

```ocaml
match [] with
  | [] -> 336
  | hd :: tl -> fold (*) (* 336 hd) tl
```
let rec fold f acc lst = match lst with
  | [] -> acc
  | hd :: tl -> fold f (f acc hd) tl

match [] with
  | [] -> 336
  | hd :: tl -> fold (*) (* 336 hd) tl
Worked Example: Product

```ocaml
let rec fold f acc lst = match lst with
| [] -> acc
| hd :: tl -> fold f (f acc hd) tl
```

336

4 Perfectly Round Circles
Insertion Sort in OCaml

```ocaml
let rec insert_sort cmp lst =
  match lst with
  | [] -> []
  | hd :: tl -> insert cmp hd (insert_sort cmp tl)
and insert cmp elt lst =
  match lst with
  | [] -> [elt]
  | hd :: tl when cmp hd elt ->
    hd :: (insert cmp elt tl)
  | _ -> elt :: lst
```

What's the worst case running time?
Sorting Examples

- langs = [“fortran”; “algol”; “c” ]
- courses = [216; 333; 415]

- sort (fun a b -> a < b) langs
  - [“algol”; “c”; “fortran” ]

- sort (fun a b -> a > b) langs
  - [“fortran”; “c”; “algol” ]

- sort (fun a b -> strlen a < strlen b) langs
  - [“c”; “algol”; “fortran” ]

- sort (fun a b -> match is_odd a, is_odd b with
  | true, false -> true (* odd numbers first *)
  | false, true -> false (* even numbers last *)
  | _, _ -> a < b (* otherwise ascending *)) courses
  - [333; 415; 216 ]
• ML, Haskell, Python, JavaScript, and Ruby all support functional programming
  - closures, anonymous functions, etc.
• ML and Haskell have strong static typing and type inference
• The others have “strong” dynamic typing (or duck typing)
• All combine OO and Functional
  - ... although it is rare to use both.
Modern Languages

• This is the most widely-spoken first language in the European Union. It is the third-most taught foreign language in the English-speaking world, after French and Spanish. Its word order is a bit more relaxed than English (since nouns are inflected to indicate their cases, as in Latin) - infamously, verbs often appear at the very end of a subordinate clause. The language's famous “Storm and Stress” movement produced classics such as Faust.
Natural Languages

• This linguist and cognitive scientist is famous for, among other things, the sentence “Colorless green ideas sleep furiously”. Introduced in his 1957 work *Syntactic Structures*, the sentence is correct but has not understandable meaning, thus demonstrating the distinction between syntax and semantics. Compare “Time flies like an arrow; fruit flies like a banana.” which illustrates garden path syntactic ambiguity.
Cool Overview

• Classroom Object-Oriented Language
• Design to
  - Be implementable in one semester
  - Give a taste of implementing modern features
    • Abstraction
    • Static Typing
    • Inheritance
    • Dynamic Dispatch
    • And more …
  - But many “grungy” things are left out
A Simple Example

```java
class Point {
    x : Int <- 0;
    y : Int <- 0;
};
```

- Cool programs are sets of class definitions
  - A special **Main** class with a special method **main**
  - Like Java
- **class** = a collection of fields and methods
- Instances of a class are **objects**
Cool Objects

class Point {
    x : Int <- 0;
    y : Int; (* use default value *)
};

- The expression “new Point” creates a new object of class Point
- An object can be thought of as a record with a slot for each attribute (= field)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Methods

A class can also define methods for manipulating its attributes

Methods refer to the current object using `self`
Aside: Semicolons

class Point {
  x : Int <- 0;
  y : Int <- 0;
  movePoint(newx : Int, newy : Int) : Point {
    { x <- newx;
      y <- newy;
      self;
    } -- close block expression
  };
  -- close method
}; -- close class

Yes, it's somewhat arbitrary. Still, don't get it wrong.
Information Hiding

• Methods are **global**
• Attributes are **local** to a class
  - They can *only* be accessed by *that class's methods*

```java
class Point {
    x : Int <- 0;
    y : Int <- 0;
    getx () : Int { x };
    setx (newx : Int) : Int { x <- newx };
}
```
Methods and Object Layout

• Each object knows how to access the code of its methods
• As if the object contains a slot pointing to the code

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>getx</th>
<th>setx</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

• In reality, implementations save space by sharing these pointers among instances of the same class

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>*</td>
</tr>
</tbody>
</table>

getx
setx
Inheritance

• We can extend points to color points using subclassing => class hierarchy

class ColorPoint extends Point {
  color : Int <- 0;
  movePoint(newx:Int, newy:Int) : Point {
    {  
      color <- 0;
      x <- newx; y <- newy;
      self;
    }
  }
};

x y color movePoint
0 0 0 *
Kool Types

• Every class is a **type**
• Base (built-in, predefined) classes:
  - **Int** for integers
  - **Bool** for booleans: true, false
  - **String** for strings
  - **Object** root of class hierarchy
• All variables must be declared
  - compiler infers types for expressions (like Java)
Cool Type Checking

- `x : Point;`
- `x <- new ColorPoint;`

• … is well-typed if **Point** is an ancestor of **ColorPoint** in the class hierarchy
  - Anywhere a **Point** is expected, a **ColorPoint** can be used (Liskov, …)

• Rephrase: … is well-typed if **ColorPoint** is a **subtype** of **Point**

• **Type safety**: a well-typed program cannot result in run-time type errors
Method Invocation and Inheritance

- Methods are invoked by (dynamic) dispatch
- Understanding dispatch in the presence of inheritance is a subtle aspect of OO
  - \( p : \text{Point}; \)
  - \( p \leftarrow \text{new ColorPoint}; \)
  - \( p.\text{movePoint}(1,2); \)
- \( p \) has static type \( \text{Point} \)
- \( p \) has dynamic type \( \text{ColorPoint} \)
- \( p.\text{movePoint} \) must invoke \( \text{ColorPoint} \) version
Other Expressions

• Cool is an expression language (like Ocaml)
  - Every expression has a type and a value
  - Conditionals if E then E else E fi
  - Loops while E loop E pool
  - Case/Switch case E of x : Type => E ; … esac
  - Assignment x <- E
  - Primitive I/O out_string(E), in_string(), …
  - Arithmetic, Logic Operations, …

• Missing: arrays, floats, interfaces, exceptions
  - Plus: you tell me!
Cool Memory Management

• Memory is allocated every time “new E” executes

• Memory is deallocated automatically when an object is not reachable anymore
  - Done by a garbage collector (GC)
Course Project

• A complete **interpreter**
  - Cool Source ==> Executed Program
  - No optimizations
  - Also no GC

• Split in 4 programming assignments (PAs)

• There is adequate time to complete assignments
  - But start early and follow directions

• PA2-5 ==> individual or teams (of max 2)

• (Compilers: Also alone or teams of two.)
Real-Time OCaml Demo

• I will code up these, with explanations, until time runs out.
  - Read in a list of integers and print the sum of all of the odd inputs.
  - Read in a list of integers and determine if any sublist of that input sums to zero.
  - Read in a directed graph and determine if node END is reachable from node START.

• You pick the order.

• Bonus: Asymptotic running times?
Homework

• PA1 Due Monday
• Reading: Chapters 2.1 - 2.2, Dijkstra, Landin

• Bonus for getting this far: questions about fold are very popular on tests! If I say “write me a function that does foozle to a list”, you should be able to code it up with fold.