Functional Programming
Introduction To Cool
Cunning Plan

• **ML Functional Programming**
  - Fold
  - Sorting
• **Cool Overview**
  - Syntax
  - Objects
  - Methods
  - Types
One-Slide Summary

• In **functional programming**, functions are first-class citizens that operate on, and produce, immutable data.

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

• Ocaml and Haskell (and Cool) support **pattern matching** over user-defined data types.

• **fold** is a powerful and general higher-order function. It can simulate many others.

• **Cool** is an object-oriented language with enough features to be indicative of modern practice.
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.
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(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)

```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 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
Recall: Fold

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

- Imagine we’re summing a list (f = addition):

  \[ 9 \rightarrow 2 \rightarrow 7 \rightarrow 4 \rightarrow 5 \rightarrow \{ \} \quad \cdots \quad \begin{array}{c}
  \text{acc} \\
  11 \end{array} \begin{array}{c}
  \text{lst} \\
  7 \rightarrow 4 \rightarrow 5 \rightarrow \{ \}
  \end{array} \]

  \[ \begin{array}{c}
  f \\
  18 \end{array} \begin{array}{c}
  \text{acc} \\
  27 \end{array} \begin{array}{c}
  \text{lst} \\
  4 \rightarrow 5 \rightarrow \{ \} \quad \cdots
  \end{array} \]
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:

  ```
  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

```ml
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

```
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

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

```ocaml
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
```
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]
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]
Worked Example: Product

\[
\text{let rec fold } f \text{ acc lst } = \text{ match lst with}
\]
\[
\text{ | } [] \rightarrow \text{ acc}
\]
\[
\text{ | } \text{hd :: tl} \rightarrow \text{ fold } f (f \text{ acc } \text{hd}) \text{ tl}
\]

\[
\text{fold } (\ast) \ 8 \ [6;7]
\]

match lst with
\[
\text{ | } [] \rightarrow \text{ acc}
\]
\[
\text{ | } \text{hd :: tl} \rightarrow \text{ fold } f (f \text{ acc } \text{hd}) \text{ tl}
\]

with \( f=\ast \), acc=8, and \( \text{lst}=[6;7] \)
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 [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

```ocaml
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

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 (*) 48 [7]

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

with f=*, acc=48, and lst=[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]
```

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

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

```ml
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
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
```
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

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
```

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

\[
\text{let rec } \text{fold } f \text{ acc } \text{lst } = \text{match lst with}
\]
\[
| \text{[]} \rightarrow \text{acc}
\]
\[
| \text{hd :: tl } \rightarrow \text{fold } f \ (f \ \text{acc} \ \text{hd}) \ \text{tl}
\]
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]
• \texttt{sort (fun a b -> a < b)} langs  
  - [ “algol”; “c”; “fortran” ]
• \texttt{sort (fun a b -> a > b)} langs  
  - [ “fortran”; “c”; “algol” ]
• \texttt{sort (fun a b -> strlen a < strlen b)} langs  
  - [ “c”; “algol”; “fortran” ]
• \texttt{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 ]

Java uses \textit{Inner Classes} for this.
• 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 no 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

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)
Methods

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

• 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>
```

```
Inheritance

• We can extend points to color points using **subclassing** \(\rightarrow\) **class hierarchy**

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

Note references to fields `x y` – They're defined in Point!
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

- \( x : \text{Point}; \)
- \( x \leftarrow \text{new ColorPoint}; \)

• ... is well-typed if \textbf{Point} is an ancestor of \textbf{ColorPoint} in the class hierarchy
  - Anywhere a \textbf{Point} is expected, a \textbf{ColorPoint} can be used (Liskov, ...)

• Rephrase: ... is well-typed if \textbf{ColorPoint} is a \textcolor{red}{subtype} of \textbf{Point}

• \textbf{Type safety}: a well-typed program \textit{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 : Point;
  - p <- new ColorPoint;
  - p.movePoint(1,2);
- p has **static** type Point
- p has **dynamic** type ColorPoint
- p.movePoint must invoke 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 garbage collection, arrays, etc.

• 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

• PA1c Due at 11:50pm ("Midnight")
  - Don't sweat it!
• Reading: Chapters 2.1 - 2.2, On-Line

• 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.