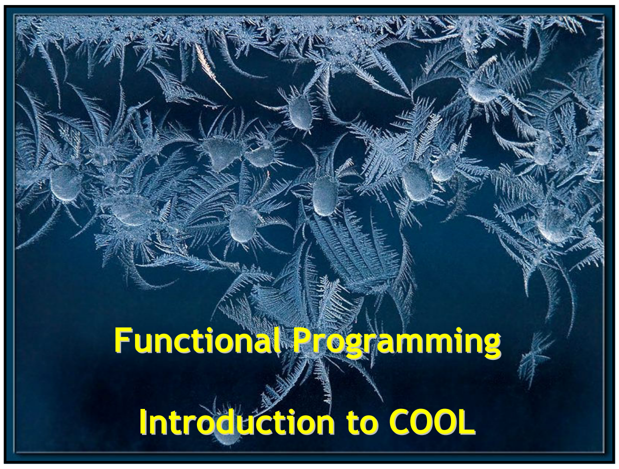
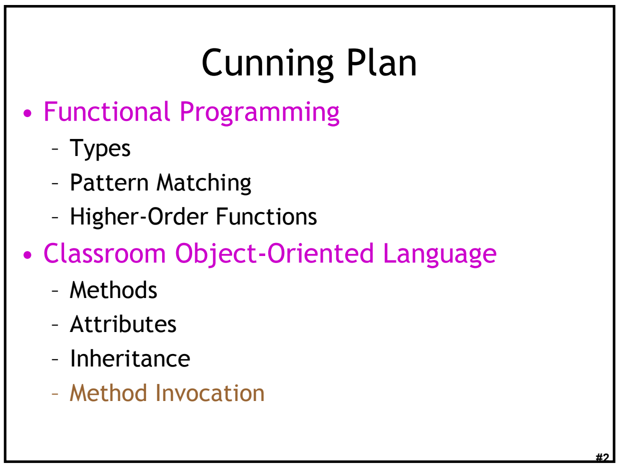


Functional Programming

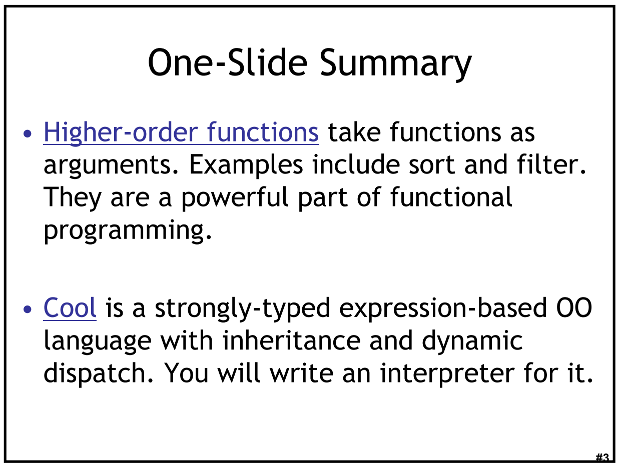
Introduction to COOL



Cunning Plan

- Functional Programming
  - Types
  - Pattern Matching
  - Higher-Order Functions
- Classroom Object-Oriented Language
  - Methods
  - Attributes
  - Inheritance
  - Method Invocation

#2



One-Slide Summary

- Higher-order functions take functions as arguments. Examples include sort and filter. They are a powerful part of functional programming.
- Cool is a strongly-typed expression-based OO language with inheritance and dynamic dispatch. You will write an interpreter for it.

#3

ML Innovative Features

- Type system
  - Strongly typed
  - Type inference
  - Abstraction
- Modules
- Patterns
- Polymorphism
- Higher-order functions
- Concise formal semantics

There are many ways of trying to understand programs. People often rely too much on one way, which is called "debugging" and consists of running a partly-understood program to see if it does what you expected. Another way, which ML advocates, is to install some means of understanding in the very programs themselves.
- **Robin Milner**, 1997

#4

Type System

- Type Inference
```
let rec add_elem (s,e) = match s with
  | [] -> [e]
  | hd :: tl when e = hd -> s
  | hd :: tl -> hd :: add_elem(tl, e)
val add_elem : α list * α -> α list = <fun>
```
- ML infers types
  - Inconsistent or incomplete type is an error
- Optional type declarations **(exp : type)**
  - Clarify ambiguous cases
  - Documentation

#5

Pattern Matching

- Simplifies Code (eliminates ifs, accessors)
```
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
```

#6

Pattern Matching Mistakes

- What if I forget a case?
  - let rec is_odd x = match x with
  - | 0 -> false
  - | 2 -> false
  - | x when x > 2 -> is_odd (x-2)
  - **Warning P: this pattern-matching is not exhaustive.**
  - **Here is an example of a value that is not matched: 1**

#7

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 : α list -> int = <fun>
  - length [1;2;3] = 3
  - length [“algol”; ”smalltalk”; ”ml”] = 3
  - length [1 ; “algol”] = ?

α means “any one type”

#8

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 = <fun>
  - let offset = 10 in
  - let myfun x = x + offset in
  - val myfun : int -> int = <fun>
  - map myfun [1;8;22] = [11;18;32]
- Extremely powerful programming technique
  - General iterators
  - Implement abstraction

f is itself a function!

#9

The Story of Fold

- We've seen **length** and **map**
- We can also imagine …

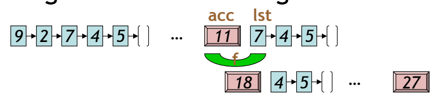
- **sum**	[1; 5; 8]	= 14
- **product**	[1; 5; 8]	= 40
- **and**	[true; true; false]	= false
- **or**	[true; true; false]	= true
- **filter**	(fun x -> x>4) [1; 5; 8]	= [5; 8]
- **reverse**	[1; 5; 8]	= [8; 5; 1]
- **mem**	5 [1; 5; 8]	= true

- Can we build all of these?

#10

The House That Fold Built

- The **fold** operator comes from Recursion Theory (Kleene, 1952)
  - 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 = <fun>**
- Imagine we're summing a list:



#11

It's Lego Time

- Let's build things out of Fold
  - **length** lst = fold (fun acc elt -> acc + 1) 0 lst
  - **sum** lst = fold (fun acc elt -> acc + elt) 0 lst
  - **product** lst = fold (fun acc elt -> acc * elt) 1 lst
  - **and** lst = fold (fun acc elt -> acc && elt) true lst
- How would we do **or**?
- How would we do **reverse**?

#12

Tougher Legos

- Examples:
  - reverse lst = fold (fun acc e -> acc @ [e]) [] lst
    - Note typing: (acc : α list) (e : α)
  - filter keep_it lst = fold (fun acc elt ->
  - if keep_it elt then elt ::acc else acc) [] lst
  - mem wanted lst = fold (fun acc elt ->
  - acc || wanted = elt) false lst
    - Note typing: (acc : bool) (e : α)
- How do we do map?
  - Recall: map (fun x -> x +10) [1;2] = [11;12]
  - Let's write it on the board …

#13

Map From Fold

- let map myfun lst =
- fold (fun acc elt -> (myfun elt) :: acc) [] lst
  - Types: (myfun : α -> β)
  - Types: (lst : α list)
  - Types: (acc : β list)
  - Types: (elt : α)
- How do we do sort?
  - (sort : (α * α -> bool) -> α list -> α list)

Do nothing which is of no use.
- **Miyamoto Musashi**, 1584-1645

#14

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]

Java uses Inner Classes for this.

#15

Partial Application and Currying

- let myadd x y = x + y
- **val myadd : int -> int -> int = <fun>**
- myadd 3 5 = 8
- let addtwo = myadd 2
  - How do we know what this means? We use referentail transparency! Basically, just sustitute it in.
- **val addtwo : int -> int = <fun>**
- addtwo 77 = 79
- Currying: “if you fix some arguments, you get a function of the remaining arguments”

#16

Applicability

- ML, Python and Ruby all support functional programming
  - closures, anonymous functions, etc.
- ML has strong static typing and type inference (as in this lecture)
- Ruby and Python have “strong” dynamic typing (or duck typing)
- All three combine OO and Functional
  - … although it is rare to use both.

#17

Q: General (458 / 842)

- This cultural anthropologist wrote the controversial 1928 classic **Coming of Age in Samoa**. The book's forward by Franz Boas includes: *"Courtesy, modesty, good manners, conformity to definite ethical standards are universal, but what constitutes courtesy, modesty, good manners, and definite ethical standards is not universal. It is instructive to know that standards differ in the most unexpected ways."*

Cool Overview

- Classroom Object Oriented Language
- Designed to
  - Be implementable in one semester
  - Give a taste of implementation of modern features
    - Abstraction
    - Static typing
    - Reuse (inheritance)
    - Memory management
    - And more …
- But many things are left out

#19

A Simple Example

```
class Point {
      x : Int ← 0;
      y : Int ← 0;
};
```

- Cool programs are sets of class definitions
  - A special class Main with a special method main
  - No separate notion of subroutine
- class = a collection of attributes and methods
- Instances of a class are objects

#20

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

x	y
0	0

#21

Methods

- A class can also define methods for manipulating the attributes

```
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
```

- Methods refer to the current object using **self**

#22

Information Hiding in Cool

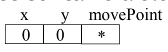
- Methods are global
- Attributes are local to a class
  - They can only be accessed by the class's methods
- Example:

```
class Point {
        . . .
        x () : Int { x };
        setx (newx : Int) : Int { x ← newx };
};
```

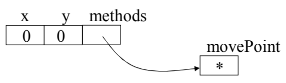
#23

Methods

- Each object knows how to access the code of a method
- As if the object contains a slot pointing to the code



- In reality implementations save space by sharing these pointers among instances of the class

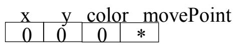


#24

Inheritance

- We can extend points to colored points using subclassing => class hierarchy

```
class ColorPoint inherits Point {
   color : Int ← 0;
   movePoint(newx : Int, newy : Int): Point {
     { color ← 0;
       x ← newx; y ← newy;
       self;
     }
   };
};
```



#25

Cool Types

- Every class is a type
- Base classes:
  - Int — for integers
  - Bool — for boolean values: true, false
  - String — for strings
  - Object — root of the class hierarchy
- All variables must be declared
  - compiler infers types for expressions (like Java)

#26

Cool Type Checking

```
x : P;
x ← new C;
```

- Is well-typed if P is an ancestor of C in the class hierarchy
  - Anywhere an P is expected a C can be used
- Type safety:
  - A well-typed program *cannot* result in runtime type errors

#27

Method Invocation and Inheritance

- Methods are invoked by (dynamic) dispatch
- Understanding dispatch in the presence of inheritance is a subtle aspect of OO languages

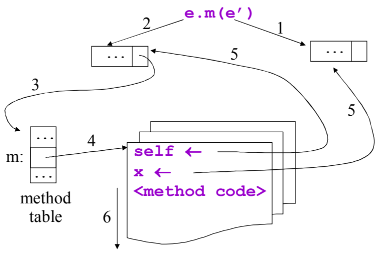
```
p : Point;
p ← new ColorPoint;
p.movePoint(1,2);
```

  - p has static type Point
  - p has dynamic type ColorPoint
  - p.movePoint must invoke the ColorPoint version

#28

Method Invocation

- Example: invoke one-argument method m



1. Eval. argum e'
2. Eval. e
3. Find class of e
4. Find code of m
5. Bind self and x
6. Run method

#29

Other Expressions

- Expression language (every expression has a type and a value)
  - Conditionals	if E then E else E fi
  - Loops:	while E loop E pool
  - Case statement	case E of x : Type ⇒ E; … esac
  - Arithmetic, logical operations
  - Assignment	x ← E
  - Primitive I/O	out_string(s), in_string(), …
- Missing features:
  - Arrays, Floating point operations, Interfaces, Exceptions, … (you tell me!)

#30

Cool Memory Management

- Memory is allocated every time new is invoked
- Memory is deallocated automatically when an object is not reachable anymore
  - Done by a garbage collector (GC)

#31

Course Project

- A complete interpreter
  - Cool Source ==> Executed Program
  - No optimizations
- Split in 5 programming assignments (PAs)
- There is adequate time to complete assignments
  - But *start early* and please follow directions
  - Turn in early to test the turn-in procedure
- Individual or team (max. 2 students)

#32

Homework

- Friday: PA0 due
- Tuesday: Chapters 2.1 - 2.2
- Tuesday: Dijkstra paper (optional)
- Tuesday: Landin paper (very optional)

#33