Functional Programming (2/2) and Intro to Cool

Martin Kellogg

Today's Agenda

- Finish introduction to functional programming
  - **polymorphism**
  - higher-order functions
    - fold
    - sorting
- Introduction to Cool
  - syntax
  - objects
  - methods
  - types

Polymorphism

- Functions and type inference in ML are ***polymorphic***

Polymorphism

- Functions and type inference in ML are ***polymorphic***
  - "Polymorphic" means they operate on more than one type

Polymorphism

- Functions and type inference in ML are ***polymorphic***
  - “Polymorphic” means they operate on more than one type

```
let rec length x = match x with
  | [] -> 0
  | hd :: tl -> 1 + length tl
```

Polymorphism

- Functions and type inference in ML are ***polymorphic***
  - "Polymorphic" means they operate on more than one type

```
let rec length x = match x with
  | [] -> 0
  | hd :: tl -> 1 + length tl

val length : α list -> int
```

Polymorphism

- Functions and type inference in ML are ***polymorphic***
  - "Polymorphic" means they operate on more than one type

```
let rec length x = match x with
  | [] -> 0
  | hd :: tl -> 1 + length tl

val length : α list -> int
```

Recall that α means "any one type"

Polymorphism

- Functions and type inference in ML are ***polymorphic***
  - "Polymorphic" means they operate on more than one type

```
let rec length x = match x with
  | [] -> 0
  | hd :: tl -> 1 + length tl

val length : α list -> int

length [1;2;3] = 3
```

Recall that α means "any one type"

Polymorphism

- Functions and type inference in ML are ***polymorphic***
  - "Polymorphic" means they operate on more than one type

```
let rec length x = match x with
  | [] -> 0
  | hd :: tl -> 1 + length tl

val length : α list -> int

length [1;2;3] = 3
length [“algol”; ”smalltalk”; ”ml”] = 3
```

Recall that α means "any one type"

Polymorphism

- Functions and type inference in ML are ***polymorphic***
  - "Polymorphic" means they operate on more than one type

```
let rec length x = match x with
  | [] -> 0
  | hd :: tl -> 1 + length tl

val length : α list -> int

length [1;2;3] = 3
length ["algol"; "smalltalk"; "ml"] = 3
length [1 ; "algol" ] = ?
```

Recall that α means "any one type"

Polymorphism

- Functions and type inference in ML are ***polymorphic***
  - “Polymorphic” means they operate on more than one type

```
let rec length x = match x with
  | [] -> 0
  | hd :: tl -> 1 + length tl
```

Recall that α means
“any one type”

```
File "list-example.ml", line 1, characters 25-26:
1 | let myList = [ "algol" ; 1 ] in
                             ^
Error: This expression has type int but an expression was expected of type
         string
```

```
length [1 ; “algol” ] = ?
```

Higher-order functions

- Functions are first-class values

Higher-order functions

- Functions are first-class values
  - Can be used whenever a value is expected (i.e., as an *expression*)

Higher-order functions

- Functions are first-class values
  - Can be used whenever a value is expected (i.e., as an *expression*)
  - Notably, can be passed around

Higher-order functions

- Functions are first-class values
  - Can be used whenever a value is expected (i.e., as an *expression*)
  - Notably, can be passed around
  - Closure captures the environment

Higher-order functions

- Functions are first-class values
  - Can be used whenever a value is expected (i.e., as an *expression*)
  - Notably, can be passed around
  - Closure captures the environment

```
let rec map f lst = match lst with
  | [] -> []
  | hd :: tl -> f hd :: map f tl
```

Higher-order functions

- Functions are first-class values
  - Can be used whenever a value is expected (i.e., as an *expression*)
  - Notably, can be passed around
  - Closure captures the environment

```
let rec map f lst = match lst with
  | [] -> []
  | hd :: tl -> f hd :: map f tl
```

f is itself a function!

Higher-order functions

- Functions are first-class values
  - Can be used whenever a value is expected (i.e., as an *expression*)
  - 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 : ?
```

Higher-order functions

- Functions are first-class values
    - Can be used whenever a value is expected (i.e., as an *expression*)
    - 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) -> ?
```

Higher-order functions

- Functions are first-class values
  - Can be used whenever a value is expected (i.e., as an *expression*)
  - 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 -> ?
```

Higher-order functions

- Functions are first-class values
  - Can be used whenever a value is expected (i.e., as an *expression*)
  - 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
```

Higher-order functions

- Functions are first-class values
  - Can be used whenever a value is expected (i.e., as an *expression*)
  - 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 : ?
```

Higher-order functions

- Functions are first-class values
  - Can be used whenever a value is expected (i.e., as an *expression*)
  - 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
```

Higher-order functions

- Functions are first-class values
  - Can be used whenever a value is expected (i.e., as an *expression*)
  - 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] = ?
```

Higher-order functions

- Functions are first-class values
  - Can be used whenever a value is expected (i.e., as an *expression*)
  - 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]
```

Higher-order functions

- Functions are first-class values
  - Can be used whenever a value is expected (i.e., as an *expression*)
  - 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

The Story of Fold

- We’ve seen **length** and **map**
- We can also imagine:

The Story of Fold

- We've seen **length** and **map**
- We can also imagine:
  - **sum** [1; 5; 8] = 14

The Story of Fold

- We've seen **length** and **map**
- We can also imagine:
  - **sum** [1; 5; 8] = 14
  - **product** [1; 5; 8] = 40

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

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

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]

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]

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

The Story of Fold

How can we **build** all of these?

- 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

The Story of Fold

- 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
```

The Story of Fold

- 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 : ?
```

The Story of Fold

- 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
```

The Story of Fold

- 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
                 f              acc    lst      (fold f acc lst)
```

The Story of Fold

- 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
```

Note: acc type and return type are the same!

```
val fold : (α -> β -> α) -> α -> β list -> α
                 f              acc    lst     (fold f acc lst)
```

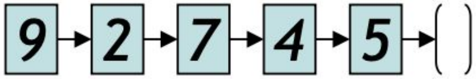
The Story of Fold

- 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
                 f               acc    lst     (fold f acc lst)
```

- on the whiteboard, this example (f is +):



Let’s build things out of fold

- **length** lst = fold (fun acc elt -> ???) ? lst

Let’s build things out of fold

- **length** lst = fold (fun acc elt -> acc + 1) 0 lst

Let's build things out of fold

- **length** lst = fold (fun acc elt -> acc + 1) 0 lst
- **sum** lst = fold (fun acc elt -> ???) ? lst

Let’s build things out of fold

- **length** lst = <u>fold</u> (fun acc elt -> acc + 1) 0 lst
- **sum** lst = <u>fold</u> (fun acc elt -> acc + elt) 0 lst

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 -> ???) ? lst

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

Let’s build things out of fold

- **length** lst = <u>fold</u> (fun acc elt -> acc + 1) 0 lst
- **sum** lst = <u>fold</u> (fun acc elt -> acc + elt) 0 lst
- **product** lst = <u>fold</u> (fun acc elt -> acc * elt) 1 lst
- **and** lst = <u>fold</u> (fun acc elt -> ???) ? lst

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

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

- think you can do **or** on your own?

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

- think you can do **or** on your own?
  - what about **reverse**?

Let’s build things out of fold, part 2

- **reverse** lst = fold (fun acc elt -> ???) ? lst

Let’s build things out of fold, part 2

- **reverse** lst = <u>fold</u> (fun acc elt -> acc @ [e]) [] lst

Let’s build things out of fold, part 2

- **reverse** lst = <u>fold</u> (fun acc elt -> acc @ [e]) [] lst
  - note types: **(acc : α list) (e : α)**

Let’s build things out of fold, part 2

- **reverse** lst = fold (fun acc elt -> acc @ [e]) [] lst
  - note types: **(acc : α list) (e : α)**
- **filter** keep_it lst = fold (fun acc elt -> ???) ? lst

Let's build things out of fold, part 2

- **reverse** lst = <u>fold</u> (fun acc elt -> acc @ [e]) [] lst
  - note types: **(acc : α list) (e : α)**
- **filter** keep_it lst = <u>fold</u> (fun acc elt -> if keep_it elt
  then elt :: acc
  else acc) [] lst

Let’s build things out of fold, part 2

- **reverse** lst = fold (fun acc elt -> acc @ [e]) [] lst
  - note types: (acc : α list) (e : α)
- **filter** keep_it lst = fold (fun acc elt -> if keep_it elt
  then elt :: acc
  else acc) [] lst
- **filter** wanted lst = fold (fun acc elt -> ???) ? lst

Let’s build things out of fold, part 2

- **reverse** lst = fold (fun acc elt -> acc @ [e]) [] lst
  - note types: (acc : α list) (e : α)
- **filter** keep_it lst = fold (fun acc elt -> if keep_it elt
  then elt :: acc
  else acc) [] lst
- **filter** wanted lst = fold (fun acc elt -> acc || wanted = elt) false lst

Let’s build things out of fold, part 2

- **reverse** lst = <u>fold</u> (fun acc elt -> acc @ [e]) [] lst
  - note types: (acc : α list) (e : α)
- **filter** keep_it lst = <u>fold</u> (fun acc elt -> if keep_it elt
  then elt :: acc
  else acc) [] lst
- **filter** wanted lst = <u>fold</u> (fun acc elt -> acc || wanted = elt) false lst
  - note types: (acc : bool) (e : α)

Let’s build things out of fold, part 2

- **reverse** lst = <u>fold</u> (fun acc elt -> acc @ [e]) [] lst
  - note types: **(acc : α list) (e : α)**
- **filter** keep_it lst = <u>fold</u> (fun acc elt -> if keep_it elt
  then elt :: acc
  else acc) [] lst
- **filter** wanted lst = <u>fold</u> (fun acc elt -> acc || wanted = elt) false lst
  - note types: **(acc : bool) (e : α)**
- Could we do **map**?
  - Recall: map (fun x -> x +10) [1;2] = [11;12]

Let's build things out of fold, part 2

- **reverse** lst = <u>fold</u> (fun acc elt -> acc @ [e]) [] lst
  - note types: **(acc : α list) (e : α)**
- **filter** keep_it lst = <u>fold</u> (fun acc elt -> if keep_it elt
  then elt :: acc
  else acc) [] lst
- **filter** wanted lst = <u>fold</u> (fun acc elt -> acc || wanted = elt) false lst
  - note types: **(acc : bool) (e : α)**
- Could we do **map**?
  - Recall: map (fun x -> x +10) [1;2] = [11;12]
  - Let's do it together…

Let's build things out of fold, part 3 (map)

```
let map myfun lst =
        fold (fun acc elt ->          ???          ) ? lst
```

Let’s build things out of fold, part 3 (map)

```
let map myfun lst =
          fold (fun acc elt -> (myfun elt) :: acc) [] lst
```

Let’s build things out of fold, part 3 (map)

```
let map myfun lst =
        fold (fun acc elt -> (myfun elt) :: acc) [] lst
```

- Types of:
  - myfun : **α -> β**
  - lst : **α list**
  - acc : **β list**
  - elt : **α**

Let’s build things out of fold, part 3 (map)

```
let map myfun lst =
        fold (fun acc elt -> (myfun elt) :: acc) [] lst
```

- Types of:
  - myfun : **α -> β**
  - lst : **α list**
  - acc : **β list**
  - elt : **α**
- Could we do **sort**?

Sorting examples

let **langs = ["fortran"; "algol"; "c"]** in

- sort (fun a b -> ???) langs = **["algol"; "c"; "fortran"]**

Sorting examples

let **langs = [“fortran”; “algol”; “c”]** in

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

Sorting examples

let **langs = [“fortran”; “algol”; “c”]** in

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

Sorting examples

let **langs = [“fortran”; “algol”; “c”]** in

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

Sorting examples

let **langs = [“fortran”; “algol”; “c”]** in

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

Sorting examples

let **langs = [“fortran”; “algol”; “c”]** in

- 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”]

Sorting examples

let **langs = [“fortran”; “algol”; “c”]** in

- 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”]

- Recall Java’s **Comparator** interface
  - in this functional style, our implementations are much simpler!

Partial Application and Currying

```
let myadd x y = x + y
val myadd : int -> int -> int
myadd 3 5 = 8
```

Partial Application and Currying

```
let myadd x y = x + y
val myadd : int -> int -> int
myadd 3 5 = 8
let addtwo = myadd 2
```

Partial Application and Currying

```
let myadd x y = x + y
val myadd : int -> int -> int
myadd 3 5 = 8
let addtwo = myadd 2
```

- How do we know what this means? We use ***referential transparency***! Basically, just substitute it in.

Partial Application and Currying

```
let myadd x y = x + y
val myadd : int -> int -> int
myadd 3 5 = 8
let addtwo = myadd 2
```

- How do we know what this means? We use ***referential transparency***! Basically, just substitute it in.

```
val addtwo : int -> int
addtwo 77 = 79
```

Partial Application and Currying

```
let myadd x y = x + y
val myadd : int -> int -> int
myadd 3 5 = 8
let addtwo = myadd 2
```

- How do we know what this means? We use ***referential transparency***! Basically, just substitute it in.

```
val addtwo : int -> int
addtwo 77 = 79
```

- called ***Currying***: “if you fix some arguments, you get a function of the remaining arguments” (remember Monday’s trivia question?)

Trivia Break: Current Events

This Chinese artificial intelligence company released a chatbot using its latest R1 model on January 10th. By January 27th, it had surpassed ChatGPT as the most-downloaded free app on the iOS App Store in the United States. The R1 model is claimed to have been trained at small fraction of the cost of training other state-of-the-art foundation models, like GPT-4.

Trivia Break: History

The New York Times mentioned this thing in 279 articles between October 6, 1957, and October 31, 1957 (more than 11 articles per day). Its launch created a crisis reaction in the United States that kicked of the "Space Race" that culminated in the Apollo moon landings in the 1960s and 1970s. After its first orbit, the Telegraph Agency of the Soviet Union (TASS) transmitted: "As result of great, intense work of scientific institutes and design bureaus the first artificial Earth satellite has been built."

Today’s Agenda

- Finish introduction to functional programming
  - polymorphism
  - higher-order functions
    - fold
    - sorting
- **Introduction to Cool**
  - syntax
  - objects
  - methods
  - types

Cool Overview

- Recall Cool = “Classroom Object-Oriented Language”
  - Designed to be implementable in one semester

Cool Overview

- Recall Cool = “Classroom Object-Oriented Language”
  - Designed to be implementable in one semester
- Give a taste of implementing modern features, such as:
  - Abstraction
  - Static Typing
  - Inheritance
  - Dynamic Dispatch
  - And more ...

Cool Overview

- Recall Cool = “Classroom Object-Oriented Language”
  - Designed to be implementable in one semester
- Give a taste of implementing modern features, such as:
  - Abstraction
  - Static Typing
  - Inheritance
  - Dynamic Dispatch
  - And more ...
- But many “grungy” things are left out

A Simple Cool Example

```
class Point {
    x : Int <- 1;
    y : Int; (* use default value *)
};
```

A Simple Cool Example

```
class Point {
    x : Int <- 1;
    y : Int; (* use default value *)
};
```

- Cool programs are sets of class definitions
  - A special `Main` class with a special method `main()`
  - Classes are like those in Java or Python or C++

A Simple Cool Example

```
class Point {
    x : Int <- 1;
    y : Int; (* use default value *)
};
```

- Cool programs are sets of class definitions
  - A special `Main` class with a special method `main()`
  - Classes are like those in Java or Python or C++
- ***class*** = a collection of fields and methods

A Simple Cool Example

```
class Point {
    x : Int <- 1;
    y : Int; (* use default value *)
};
```

- Cool programs are sets of class definitions
  - A special `Main` class with a special method `main()`
  - Classes are like those in Java or Python or C++
- ***class*** = a collection of fields and methods
- Instances of a class are ***objects***

Cool Objects

```
class Point {
    x : Int <- 1;
    y : Int; (* use default value *)
};
```

- The expression `new Point` creates a new object of class `Point`

Cool Objects

```
class Point {
    x : Int <- 1;
    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)

x	y
1	0

Cool Methods

A class can also define **methods** for manipulating its attributes

```
class Point {
    x : Int <- 1;
    y : Int; (* use default value *)
   movePoint(newx : Int, newy : Int) : Point {
            { x <- newx;
              y <- newy;
              self;
            } -- close block expression
       }; -- close method
 }; -- close class
```

Cool Methods

```
class Point {
    x : Int <- 1;
    y : Int; (* use default value *)
   movePoint(newx : Int, newy : Int) : Point {
            { x <- newx;
              y <- newy;
              self;
            } -- close block expression
        }; -- close method
 }; -- close class
```

Methods can refer to the current object using the self keyword

Cool Methods

```
class Point {
    x : Int <- 1;
    y : Int; (* use default value *)
    movePoint(newx : Int, newy : Int) : Point {
            { x <- newx;
              y <- newy;
              self;
            } -- close block expression
        }; -- close method
};  -- close class
```

Aside: yes, the placement of semicolons is arbitrary. Still, don't get it wrong.

Cool: Information Hiding

- Cool's methods are ***global***: they can be accessed from any other part of the program
  - like `public` in Java

Cool: Information Hiding

- Cool's methods are ***global***: they can be accessed from any other part of the program
  - like `public` in Java
- Attributes, on the other hand, are ***local***: they can **only** be accessed by **that class'** methods

Cool: Information Hiding

- Cool's methods are ***global***: they can be accessed from any other part of the program
  - like `public` in Java
- Attributes, on the other hand, are ***local***: they can **only** be accessed by **that class'** methods
  - conveniently, this means there is **no dereference syntax**
    - e.g., you can't write `f.x` in Cool!

Cool: Information Hiding

- Cool's methods are ***global***: they can be accessed from any other part of the program
  - like `public` in Java
- Attributes, on the other hand, are ***local***: they can **only** be accessed by **that class'** methods
  - conveniently, this means there is **no dereference syntax**
    - e.g., you can't write `f.x` in Cool!
  - instead, all attributes are accessed directly by name
    - simplifies reasoning about **scopes** (we'll come back to it)

Cool: Methods and Object Layout

- Each object knows how to access the code of its methods

Cool: 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**:

x	y	movePoint
1	0	*

(code for movePoint)

Cool: 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**:

x	y	movePoint
1	0	*

(code for movePoint)

- This is a simplification: in reality, implementations save space by **sharing these pointers** among instances of the same class:

x	y	methods
1	0	*

movePoint
setX
...

(code for movePoint)

(code for setX)

Cool: 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**:

`x`	`y`	`movePoint`
`1`	`0`	`*`

(code for movePoint)

Another topic we'll cover in a lot more detail later!

- This is a simplification: in reality, implementations save space by **sharing these pointers** among instances of the same class:

`x`	`y`	`methods`
`1`	`0`	`*`

`movePoint`
`setX`
`...`

(code for movePoint)

(code for setX)

Cool: Inheritance

- We can extend points to color points using ***subclassing***, which gives us a ***class hierarchy***. E.g.,:

Cool: Inheritance

- We can extend points to color points using ***subclassing***, which gives us a ***class hierarchy***. E.g.,:

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

Cool: Inheritance

- We can extend points to color points using ***subclassing***, which gives us a ***class hierarchy***. E.g.,:

Can add new attributes

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

Cool: Inheritance

- We can extend points to color points using ***subclassing***, which gives us a ***class hierarchy***. E.g.,:

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

Can redefine methods (= Java's *overriding*)

Cool: Inheritance

- We can extend points to color points using ***subclassing***, which gives us a ***class hierarchy***. E.g.,:

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

Can still reference attributes defined in superclass

Cool: Types

- Every class in Cool **defines** a type

Cool: Types

- Every class in Cool **defines** a type
- Base (built-in, predefined classes):
  - **Int** for integers (including all integer literals)
  - **Bool** for booleans (including `true` and `false`)
  - **String** for strings (including string literals)
  - **Object** root of class hierarchy
  - **IO** for built-in input/output operations

Cool: Types

- Every class in Cool **defines** a type
- Base (built-in, predefined classes):
  - **Int** for integers (including all integer literals)
  - **Bool** for booleans (including `true` and `false`)
  - **String** for strings (including string literals)
  - **Object** root of class hierarchy
  - **IO** for built-in input/output operations
- All variables must be **declared** with their type

Cool: Types

- Every class in Cool **defines** a type
- Base (built-in, predefined classes):
  - **Int** for integers (including all integer literals)
  - **Bool** for booleans (including `true` and `false`)
  - **String** for strings (including string literals)
  - **Object** root of class hierarchy
  - **IO** for built-in input/output operations
- All variables must be **declared** with their type
  - compiler **infers types** for expressions (like Java or C)

Cool: Typechecking

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

Cool: Typechecking

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

- This program is **well-typed** iff Point is an ancestor of ColorPoint in the type hierarchy

Cool: Typechecking

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

- This program is **well-typed** iff Point is an ancestor of ColorPoint in the type hierarchy
  - And anywhere a Point is expected, we can use a ColorPoint

Cool: Typechecking

This is called the ***Liskov Substitution Principle***: "any subclass object should be safe to use in place of a superclass object at run time"

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

- This program is **well-typed** iff Point is an ancestor of ColorPoint in the type hierarchy
  - And anywhere a Point is expected, we can use a ColorPoint

Cool: Typechecking

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

This is called the ***Liskov Substitution Principle***: “any subclass object should be safe to use in place of a superclass object at run time”

- This program is **well-typed** iff Point is an ancestor of ColorPoint in the type hierarchy
  - And anywhere a Point is expected, we can use a ColorPoint
- Another way of phrasing this: … is well-typed iff ColorPoint is a ***subtype*** of Point

Cool: Typechecking

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

> This is called the ***Liskov Substitution Principle***: “any subclass object should be safe to use in place of a superclass object at run time”

- This program is **well-typed** iff Point is an ancestor of ColorPoint in the type hierarchy
  - And anywhere a Point is expected, we can use a ColorPoint
- Another way of phrasing this: … is well-typed iff ColorPoint is a ***subtype*** of Point
- Cool’s ***type safety theorem*** says that a well-typed program **cannot** result in run-time type errors

Pedantic aside: runtime vs run-time vs run time

There are three similar words that people often confuse in Computer Science:

Pedantic aside: runtime vs run-time vs run time

There are three similar words that people often confuse in Computer Science:

- a ***runtime*** is a computer program that provides an environment in which to execute other programs
  - e.g., the JVM, your favorite shell

Pedantic aside: runtime vs run-time vs run time

There are three similar words that people often confuse in Computer Science:

- a ***runtime*** is a computer program that provides an environment in which to execute other programs
  - e.g., the JVM, your favorite shell
- ***run time*** (a noun) is either the time at which a program runs or the amount of time it takes to execute

Pedantic aside: runtime vs run-time vs run time

There are three similar words that people often confuse in Computer Science:

- a ***runtime*** is a computer program that provides an environment in which to execute other programs
  - e.g., the JVM, your favorite shell
- ***run time*** (a noun) is either the time at which a program runs or the amount of time it takes to execute
- ***run-time*** (an adjective) describes something that happens at run time

Cool: Method Invocation + Inheritance

Cool: Method Invocation + Inheritance

- Methods are invoked via ***dynamic dispatch***

Cool: Method Invocation + Inheritance

- Methods are invoked via ***dynamic dispatch***
- Understanding dispatch in the presence of inheritance is a subtle aspect of object-oriented programming.

Cool: Method Invocation + Inheritance

- Methods are invoked via ***dynamic dispatch***
- Understanding dispatch in the presence of inheritance is a subtle aspect of object-oriented programming. E.g.,:

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

Cool: Method Invocation + Inheritance

- Methods are invoked via ***dynamic dispatch***
- Understanding dispatch in the presence of inheritance is a subtle aspect of object-oriented programming. E.g.,:

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

- p has static type **Point**

Cool: Method Invocation + Inheritance

- Methods are invoked via ***dynamic dispatch***
- Understanding dispatch in the presence of inheritance is a subtle aspect of object-oriented programming. E.g.,:

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

- p has static type **Point**
- p has dynamic type **ColorPoint**

Cool: Method Invocation + Inheritance

- Methods are invoked via ***dynamic dispatch***
- Understanding dispatch in the presence of inheritance is a subtle aspect of object-oriented programming. E.g.,:

```
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!

Cool: Method Invocation + Inheritance

- Methods are invoked via ***dynamic dispatch***
- Understanding dispatch in the presence of inheritance is a subtle aspect of object-oriented programming. E.g.,:

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

Aside that will come up again: *static* means “just by reading the program text”; *dynamic* means “at run time”

- p has static type **Point**
- p has dynamic type **ColorPoint**
- `p.movePoint()` **must** invoke the **ColorPoint** version!

Cool: Other Expressions

- Cool is an **expression language** (like OCaml)

Cool: Other Expressions

- Cool is an **expression language** (like OCaml)
    - every expression has both a type and a value

Cool: Other Expressions

- Cool is an **expression language** (like OCaml)
  - every expression has both a type and a value
- Includes:
  - Conditionals: if **E** then **E** else **E** fi

Cool: Other Expressions

- Cool is an expression language (like OCaml)
  - every expression has both a type and a value
- Includes:
  - Conditionals: if **E** then **E** else **E** fi
  - Loops: while **E** loop **E** pool

Cool: Other Expressions

- Cool is an **expression language** (like OCaml)
  - every expression has both a type and a value
- Includes:
  - Conditionals: if **E** then **E** else **E** fi
  - Loops: while **E** loop **E** pool
  - Case/switch: case **E** of **x** : **Type** => **E** ; … esac

Cool: Other Expressions

- Cool is an expression language (like OCaml)
  - every expression has both a type and a value
- Includes:
  - Conditionals: if **E** then **E** else **E** fi
  - Loops: while **E** loop **E** pool
  - Case/switch: case **E** of **x** : **Type** => **E** ; … esac
  - Assignments: **x** <- **E**

Cool: Other Expressions

- Cool is an expression language (like OCaml)
  - every expression has both a type and a value
- Includes:
  - Conditionals: if **E** then **E** else **E** fi
  - Loops: while **E** loop **E** pool
  - Case/switch: case **E** of **x** : **Type** => **E** ; … esac
  - Assignments: **x** <- **E**
  - Arithmetic, logic operators, comparison operators, etc.

Cool: Other Expressions

- Cool is an expression language (like OCaml)
  - every expression has both a type and a value
- Includes:
  - Conditionals: if **E** then **E** else **E** fi
  - Loops: while **E** loop **E** pool
  - Case/switch: case **E** of **x** : **Type** => **E** ; … esac
  - Assignments: **x** <- **E**
  - Arithmetic, logic operators, comparison operators, etc.
- Missing: arrays, floats, interfaces, exceptions…
  - any other missing things you noticed?

Cool: Memory Management

Cool: Memory Management

- Memory is allocated every time that `new E` executes

Cool: Memory Management

- Memory is allocated every time that `new E` executes
- Memory is deallocated **automatically** when an object is no longer reachable

Cool: Memory Management

- Memory is allocated every time that `new E` executes
- Memory is deallocated **automatically** when an object is no longer reachable
    - this is the job of a ***garbage collector***

Cool: Memory Management

- Memory is allocated every time that `new E` executes
- Memory is deallocated **automatically** when an object is no longer reachable
  - this is the job of a ***garbage collector***
  - your compiler will need to not leak **too much** memory…

Cool: Memory Management

- Memory is allocated every time that `new E` executes
- Memory is deallocated **automatically** when an object is no longer reachable
  - this is the job of a ***garbage collector***
  - your compiler will need to not leak **too much** memory…
    - …but building a good garbage collector is just **one of many** paths to high-performance assembly

Cool: Memory Management

- Memory is allocated every time that `new E` executes
- Memory is deallocated **automatically** when an object is no longer reachable
  - this is the job of a ***garbage collector***
  - your compiler will need to not leak **too much** memory…
    - …but building a good garbage collector is just **one of many** paths to high-performance assembly
  - we'll cover garbage collectors in more detail (much) later in the semester

Course Announcements

- Don't forget: PA1c2 **due tomorrow**
  - and full PA1 (all four languages!) **due next Monday**

Course Announcements

- Don’t forget: PA1c2 **due tomorrow**
  - and full PA1 (all four languages!) **due next Monday**
- My OH today are modified (conflicting CS faculty candidate meeting):
  - I will hold OH as usual from 3:30 until 4pm
  - OH will end 30 minutes early at 4pm
  - to account for this, I’ll hold extra OH from 4:30-5

Course Announcements

- Don’t forget: PA1c2 **due tomorrow**
  - and full PA1 (all four languages!) **due next Monday**
- My OH today are modified (conflicting CS faculty candidate meeting):
  - I will hold OH as usual from 3:30 until 4pm
  - OH will end 30 minutes early at 4pm
  - to account for this, I’ll hold extra OH from 4:30-5
- PA2c1 is due **next Friday** (do it next week!)
  - this is a testing assignment: you’ll just write Cool programs

Bonus for those reading the slides :)

- 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 in OCaml-ish syntax.