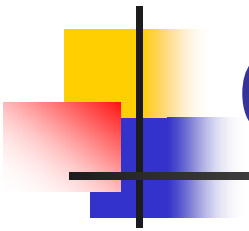
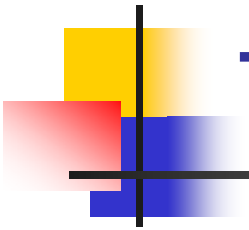


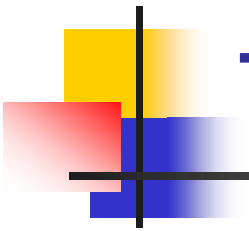
CS 11 Ocaml track: lecture 5

- Today:
  - functors



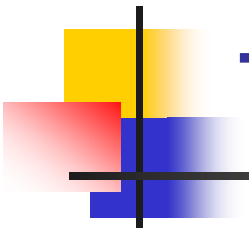
The idea of functors (1)

- Often have a situation like this:
  - You want a module type
  - It should be parameterized around some other type or types
  - But not *every* type or types
- In other words, want a *restricted* kind of polymorphism



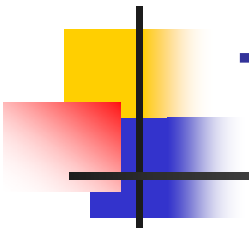
The idea of functors (2)

- In other words, want a *restricted* kind of polymorphism
- The notion of "restricted polymorphism" doesn't exist as such in ocaml
  - for that, see Haskell (type classes)
- However, can "fake it" using functors



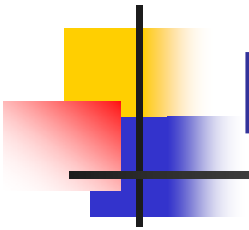
The idea of functors (3)

- Consider polymorphic types as being like a "function" on types
  - Example: type 'a list means "give me a type 'a, and I'll give you a type 'a list"
- Similarly, imagine a "function" on modules
  - Give me a module representing the type that varies (analogous to 'a) and I'll give you a module representing the type that uses it (analogous to 'a list)



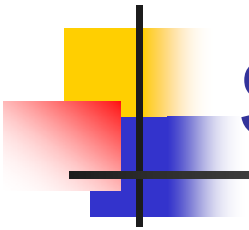
The idea of functors (4)

- Similarly, imagine a "function" on modules
  - Give me a module representing the type that varies and I'll give you a module representing the type that uses it
- These "functions on modules" are called functors in ocaml
- The name "functor" comes from category theory (not important why)



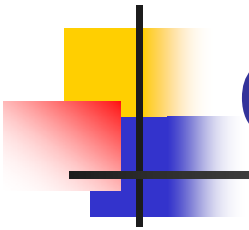
Example

- Consider the Set module we developed last lecture
- Some set implementations require set elements to be *orderable* (*i.e.* notion of "less than", "equal", "greater than" is meaningful)
- Why would you want this?



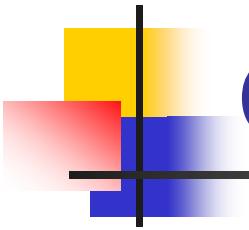
Sets with orderable elements

- Can get better efficiency with Set implementation if know that elements are orderable
- Example: Set implementation: ordered binary tree
  - left subtree: all elems < node elem
  - right subtree: all elems >= node elem
- member function now O(log n) instead of O(n)
  - HUGE win!



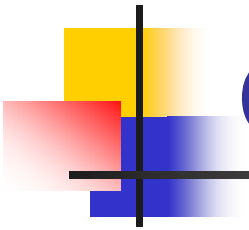
OrderedSet (1)

- We will call our set that can only work with orderable elements OrderedSet
- A large variety of types can work with OrderedSet
  - set of numbers
  - set of strings
  - set of chars
- But not *every* type
  - set of functions? (no ordering)



OrderedSet (2)

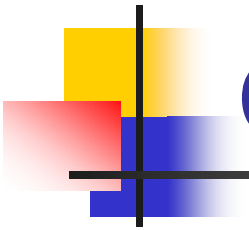
- Since cannot use any type as the element of an OrderedSet, cannot have a polymorphic type as the set element
  - i.e. can't use PolySet as defined last lecture
- Could just define a whole new OrderedSet for each type we want to make a set of
  - OrderedSetInt
  - OrderedSetString
- But wasteful, since code is nearly identical



OrderedSet elements (1)

- How do we characterize the essential nature of types that can be elements of our OrderedSet?
- We define a module type:

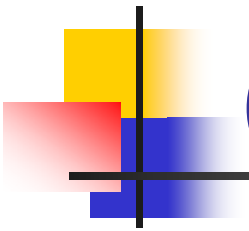
```
type comparison = Less | Equal | Greater
module type ORDERED_TYPE =
  sig
    type t  (* type of elements *)
    val compare: t -> t -> comparison
  end
```



OrderedSet elements (2)

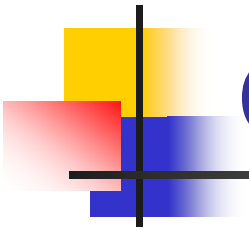
- Example of a module compatible with ORDERED_TYPE:

```
module OrderedString =
  struct
    type t = string
    let compare x y =
      if x = y then Equal
      else if x < y then Less
      else Greater
  end
```



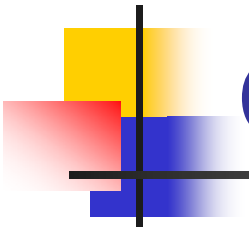
(Recall) SET module type

```
module type SET =
  sig
    type element
    type set
    val empty : set
    val add : element -> set -> set
    val member : element -> set -> bool
  end
```



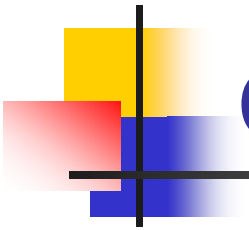
OrderedSet functor (1)

- Now, the question becomes:
- Given a module compatible with the module type ORDERED_TYPE,
- How do I create a module that is compatible with the module type SET?
- Answer: define a functor:
  - "function" that takes in a module compatible with module type ORDERED_TYPE, and
  - creates a new module, compatible with module type SET



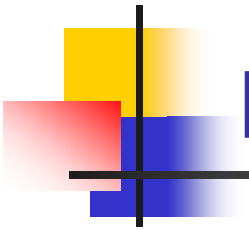
OrderedSet functor (2)

- Advantage of functor: any time need a new ORDERED_SET for a new kind of orderable type:
  - Define a new module for that type compatible with ORDERED_TYPE (easy)
  - Apply functor to the module, get new set module
- Disadvantage of functor:
  - One extra level of indirection
  - Therefore somewhat slower than definition without functor



OK, but how do we *define* functors?

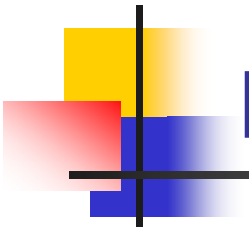
- There are several different ways to define functors
- All described (poorly) in Ocaml manual
- All basically equivalent
- IMO very messy syntactically (like rest of ocaml)
- I will show you one way that will work
  - be aware that this can be done other ways
  - see ocaml manual and Jason's book for those



Defining a functor (skeleton)

```
module MakeOrderedSet(Elt: ORDERED_TYPE) :
    (SET with type element = Elt.t) =
  struct
    (* details omitted for now *)
  end
```

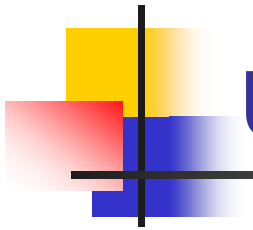
- Like a function:
  - input: a module Elt that conforms to ORDERED_TYPE
  - output: a module that conforms to SET with the type element the same as the type Elt.t



Defining a functor (alternative)

```
module MakeOrderedSet(Elt: ORDERED_TYPE) =
  struct
    (* details omitted for now *)
  end
```

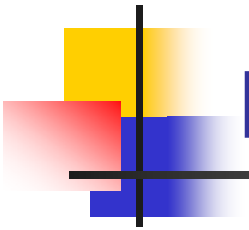
- Here we've omitted the "result module type"
- Will still work correctly
- However, resulting module will not be abstract
  - internals of module will be publicly visible (usually bad)
  - We'll stick with abstract version



Using a functor

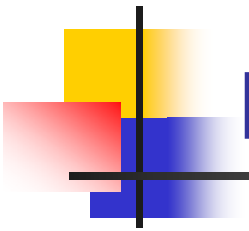
```
module OrderedStringSet = MakeOrderedSet(OrderedString);;
→ module OrderedStringSet :
    sig
      type element = OrderedString.t
      type set = MakeOrderedSet(OrderedString).set
      val empty : set
      val add : element -> set -> set
      val member : element -> set -> bool
    end

let set1 = OrderedStringSet.add "gee" OrderedStringSet.empty;;
→ val set1 : OrderedStringSet.set = <abstr>
```



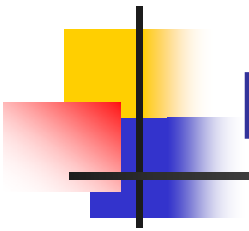
Defining a functor (details) (1)

- To simplify code, we'll define our Set functor not in terms of ordered binary trees but in terms of ordered lists
- This will not be nearly as efficient
  - *e.g.* member will still be O(N)
  - but functor concepts will be just as clear
- Binary tree version left as exercise



Defining a functor (details) (2)

```
module MakeOrderedSet(Elt: ORDERED_TYPE) :
      (SET with type element = Elt.t) =
   struct
      type element = Elt.t       (* note code duplication *)
      type set = element list
      let empty = []
      (* continued on next slide *)
```

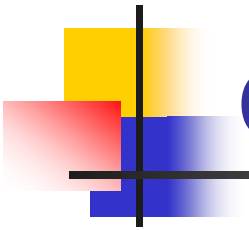


Defining a functor (details) (3)

```
(* continued from previous slide *)
let rec add x s =
   match s with
      | [] -> [x]
      | hd :: tl ->
           match Elt.compare x hd with
              | Equal   -> s
              | Less    -> x :: s
              | Greater -> hd :: add x tl
(* continued on next slide *)
```

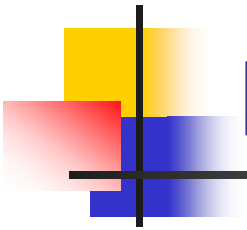
Defining a functor (details) (4)

```
    (* continued from previous slide *)
    let rec member x s =
       match s with
          | [] -> false
          | hd :: tl ->
               match Elt.compare x hd with
                  | Equal   -> true
                  | Less    -> false
                  | Greater -> member x tl
end  (* of struct *)
```



Other functor stuff

- It's possible to define multi-parameter functors
  - *i.e.* functors with several module inputs
- Syntax is gnarly (surprise, surprise)
- Probably won't need to do that



Next time

- Building a language, part 1
- Parser generators
  - ocamllex
  - ocamlyacc