Programming Languages and Techniques (CIS120)

Lecture 10

Abstract types: Sets

Chapter 10

Announcements

- Homework 3
  - due *Tuesday* at 11:59:59pm

- Reading: Chapters 8, 9, and 10 of the lecture notes

- Midterm 1: Friday, September 27th
  - Coverage: up to Monday, Sept. 23 (Chs. 1-10)
  - During lecture (001 @ 11am, 002 @ noon)
    Last names: A – L Leidy Labs 10
    Last names: M – Z Stitler (STIT) B6
  - Review Material
    - old exams on the web site lecture schedule
  - Makeup exam: Monday, Sept. 30th
    - sign up form on the web site

The ‘fold’ design pattern

List Fold

```
let rec fold (combine: 'a -> 'b -> 'b)
             (base:'b) (l : 'a list) : 'b =
   begin match l with
   | [] -> base
   | x :: t -> combine x (fold combine base t)
   end

let exists (l : bool list) : bool =
    fold (fun (h:bool) (acc:bool) -> h || acc) false l

let acid_length (l : acid list) : int =
    fold (fun (h:acid) (acc:int) -> 1 + acc) 0 l
```

- `fold` (a.k.a. "reduce")
  - Like transform, foundational function for programming with lists
  - Captures the pattern of recursion over lists
  - Also part of OCaml standard library (`List.fold_right`)
  - Similar operations for other recursive datatypes (`fold_tree`)

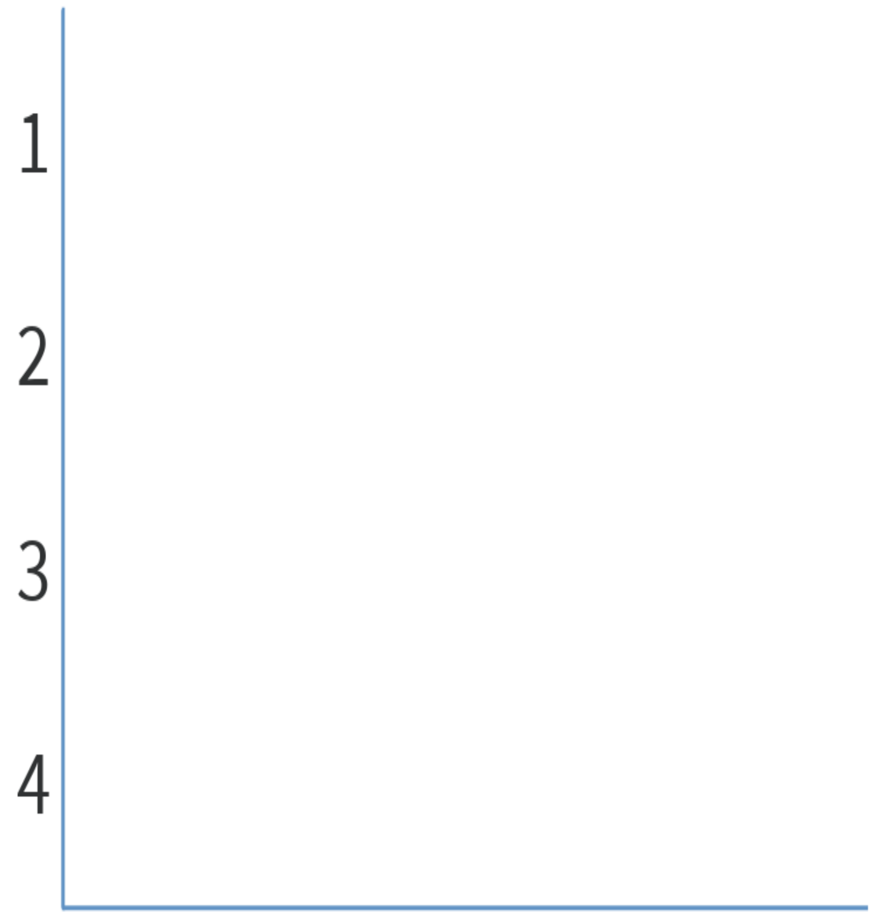
Rewrite using fold

How would you rewrite this function

```
let rec sum (l : int list) : int =
  begin match l with
  | [] -> 0
  | h :: t -> h + sum t
  end
```

using fold? What should be the arguments for base and combine?

1. combine is: `(fun (h:int) (acc:int) -> acc + 1)`
   base is: `0`
2. combine is: `(fun (h:int) (acc:int) -> h + acc)`
   base is: `0`
3. combine is: `(fun (h:int) (acc:int) -> h + acc)`
   base is: `1`
4. sum can't be written with `fold`.



How would you rewrite this function

```
let rec sum (l : int list) : int =
   begin match l with
   | [] -> 0
   | h :: t -> h + sum t
   end
```

using fold? What should be the arguments for base and combine?

1. combine is: `(fun (h:int) (acc:int) -> acc + 1)`
   base is: `0`

2. combine is: `(fun (h:int) (acc:int) -> h + acc)`
   base is: `0`

3. combine is: `(fun (h:int) (acc:int) -> h + acc)`
   base is: `1`

4. sum can't be written with `fold`.

Answer: 2

Rewrite using fold

How would you rewrite this function

```
let rec reverse (l : int list) : int list =
  begin match l with
  | [] -> []
  | h :: t -> reverse t @ [h]
  end
```

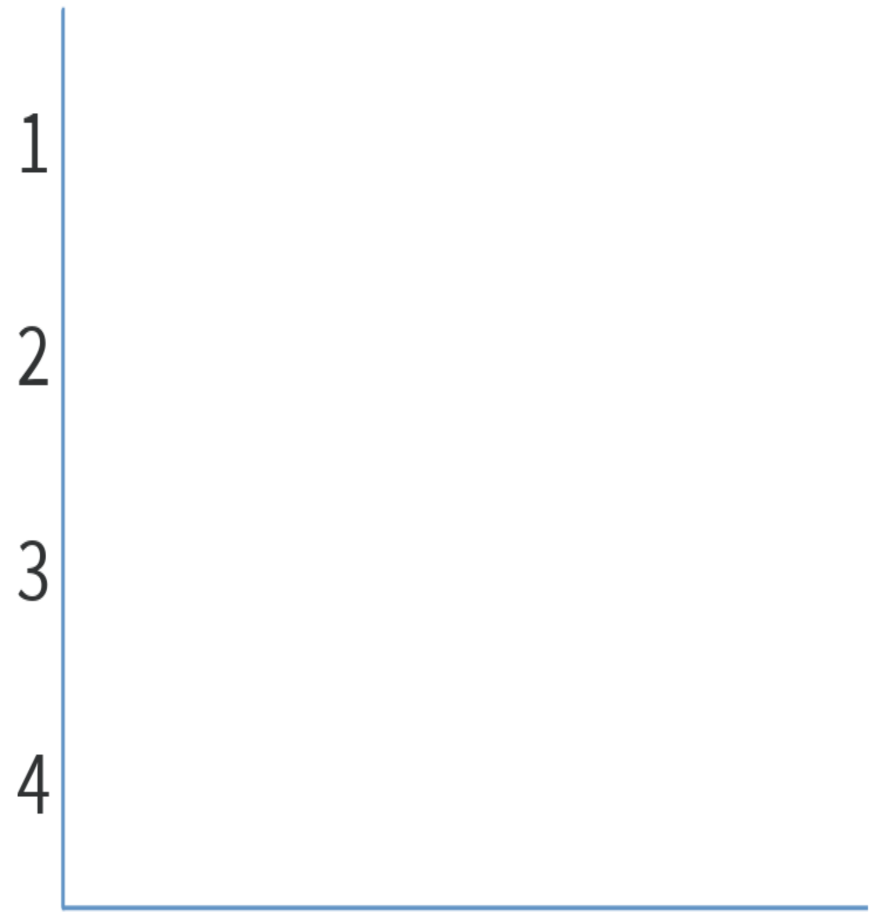
using fold? What should be the arguments for base and combine?

1. combine is: `(fun (h:int) (acc:int list) -> h :: acc)`
   base is: `0`

2. combine is: `(fun (h:int) (acc:int list) -> acc @ [h])`
   base is: `0`

3. combine is: `(fun (h:int) (acc:int list) -> acc @ [h])`
   base is: `[]`

4. reverse can't be written by with `fold`.



How would you rewrite this function

```
let rec reverse (l : int list) : int list =
   begin match l with
   | [] -> []
   | h :: t -> reverse t @ [h]
   end
```

using fold? What should be the arguments for base and combine?

1. combine is: `(fun (h:int) (acc:int list) -> h :: acc)`
   base is: `0`

2. combine is: `(fun (h:int) (acc:int list) -> acc @ [h])`
   base is: `0`

3. combine is: `(fun (h:int) (acc:int list) -> acc @ [h])`
   base is: `[]`

4. reverse can't be written by with `fold`.

Answer: 3

See hof.ml

MORE EXAMPLES OF FOLD

Functions as Data

- We've seen a number of ways in which functions can be treated as data in OCaml
- Everyday programming practice offers many more examples
  - objects bundle "functions" (a.k.a. methods) with data
  - iterators ("cursors" for walking over data structures)
  - event listeners (in GUIs)
  - etc.
- Also heavily used at "large scale": Google's MapReduce
  - Framework for transforming (mapping) sets of key-value pairs
  - Then "reducing" the results per key of the map
  - Easily distributed to 10,000 machines to execute in parallel!

Abstract Collections

Mathematical Sets

- Mathematical sets represent *collections* of things:

Empty Set:	$\emptyset$	no things
Nonempty Sets:	{0, 1, 2, 3}	four integers
	{(0,1), (2,3)}	two points in the plane
	{true, false}	two Boolean values
Manipulating Sets:		
	$S \cup T$	union
	$S \cap T$	intersection
Predicates:	$x \in S$	"x is a member of set S"

A *set* is an abstraction

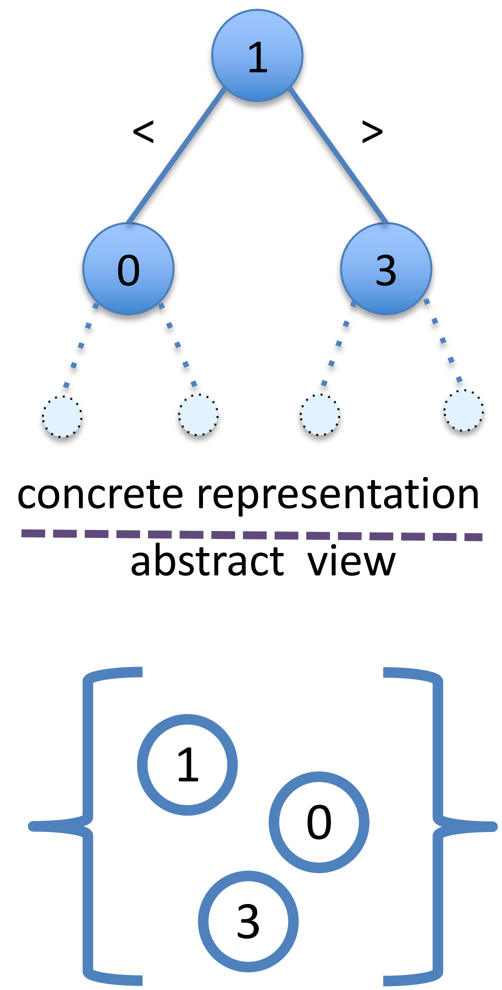
- A set is a collection of data
  - we have operations for forming sets of elements
  - we can ask whether elements are in a set
- A set is a lot like a list, except:
  - Order doesn't matter
  - Duplicates don't matter
  - *It isn't built into OCaml*

  An element's *presence* or *absence* in the set is all that matters…
- Sets show up frequently in applications
  - Examples: set of students in a class, set of coordinates in a graph, set of answers to a survey, set of data samples from an experiment, …

Abstract type: set

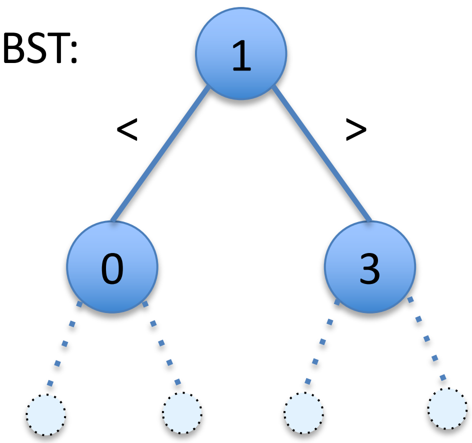
- A BST can *implement (represent)* a *set*
  - *there is a way to represent an empty set (Empty)*
  - *there is a way to list all elements contained in the set (inorder)*
  - *there is a way to test membership (lookup)*
  - *Can define union/intersection (with insert and delete)*

- *BSTs are not the only way to implement sets*



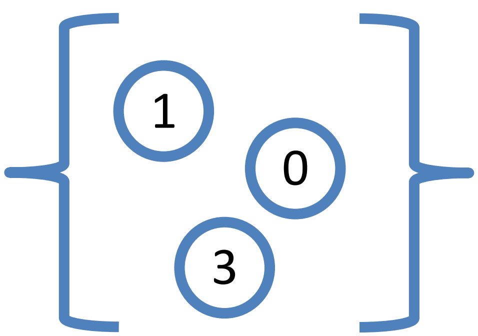
Three Example Representations of Sets

BST:



concrete representation

abstract view

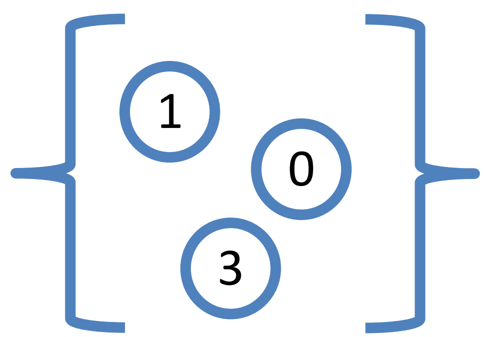


Alternate representation: unsorted linked list.

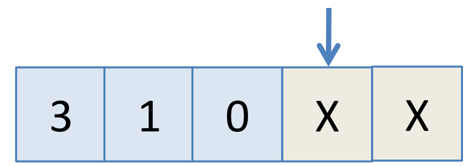
3::0::1::[]

concrete representation

abstract view

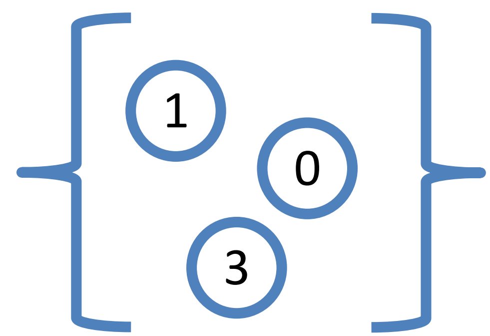


Alternate representation: reverse sorted array with Index of next slot.



concrete representation

abstract view

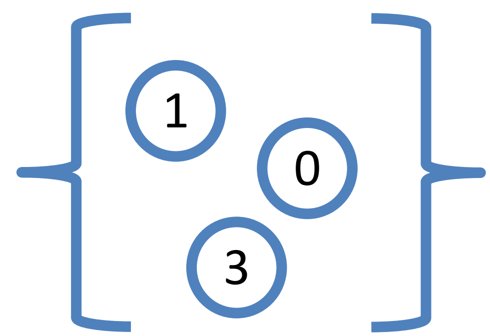


Abstract types (e.g. set)

- An abstract type is defined by its *interface* and its *properties*, not its representation
- Interface: defines the type and operations
  - There is a type of sets
  - There is an empty set
  - There is a way to add elements to a set to make a bigger set
  - There is a way to list all elements in a set
  - There is a way to test membership
- Properties: define how the operations interact with each other
  - Elements that were added can be found in the set
  - Adding an element a second time doesn't change the elements of a set
  - Adding in a different order doesn't change the elements of a set
- *Any* type that satisfies the interface and properties can be a set

concrete representation

abstract view



Sets in OCaml

OCaml directly supports the declaration of abstract types via *signatures*

Set *Signature*

The name of the signature

The `sig` keyword indicates an interface declaration

```
module type SET = sig

   type 'a set

   val empty       : 'a set
   val add         : 'a -> 'a set -> 'a set
   val member      : 'a -> 'a set -> bool
   val equals      : 'a set -> 'a set -> bool
   val set_of_list : 'a list -> 'a set

end
```

Type declaration has no "right-hand side" – its representation is *abstract*!

The interface members are the (only!) means of manipulating the abstract type.

Signature (a.k.a. interface): defines operations on the type

Implementing sets

- There are many ways to implement sets
  - lists, trees, arrays, etc.
- *How do we choose which implementation?*
  - Depends on the needs of the application…
  - How often is ‘member’ used vs. ‘add’?
  - How big can the sets be?

- Many such implementations are of the flavor
  “a set is a … with some *invariants*”
  - A set is a *list* with no repeated elements.
  - A set is a *tree* with no repeated elements
  - A set is a *binary search tree*
  - A set is an *array of bits*, where 0 = absent, 1 = present
- *How do we preserve the invariants of the implementation?*

Invariant: a property that remains unchanged when a specified transformation is applied.

A *module* implements an interface

- An implementation of the set interface will look like this:

Name of the module

Signature that it implements

The **struct** keyword indicates a module implementation

```
module BSTSet : SET = struct
  …
  (* implementations of type and operations *)
  …
end
```

Implement the `set` Module

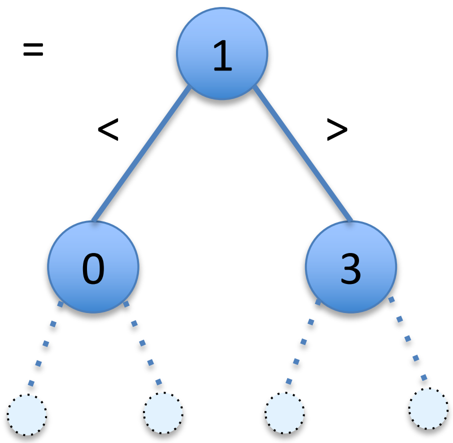
```
module BSTSet : SET = struct

  type 'a tree =
   | Empty
   | Node of 'a tree * 'a * 'a tree

  type 'a set = 'a tree

  let empty : 'a set = Empty
  …
end
```

Module must define (give a *concrete representation* to) the type declared in the signature

- The implementation must include everything promised by the interface
  - It can contain *more* functions and type definitions (e.g. auxiliary or helper functions) but those cannot be used outside the module
  - The types of the provided implementations must match the signature

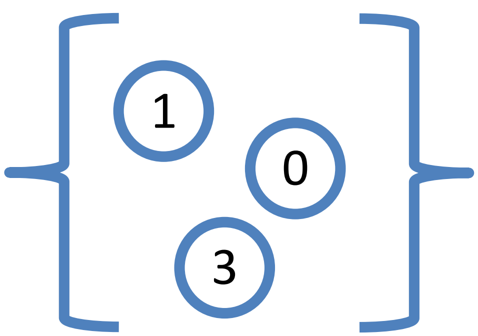
Abstract vs. Concrete BSTSet

s =



concrete representation

abstract view




```
module BSTSet : SET = struct
  type 'a tree = …
  type 'a set = 'a tree
  let empty : 'a set = Empty
  let add (x:'a)  (s:'a set) :'a set =
     ... (* can treat s as a tree *)

end
```

```
module type SET = sig
  type 'a set
  val empty : 'a set
  val add   : 'a -> 'a set -> 'a set
end
```

```
(* A client of the BSTSet module *)
;; open BSTSet

let s : int set
  = add 0 (add 3 (add 1 empty))
```

Another Implementation

```
module ULSet : SET =
struct

  type 'a set = 'a list

  let empty : 'a set = []
  …

end
```

A different definition for the type set

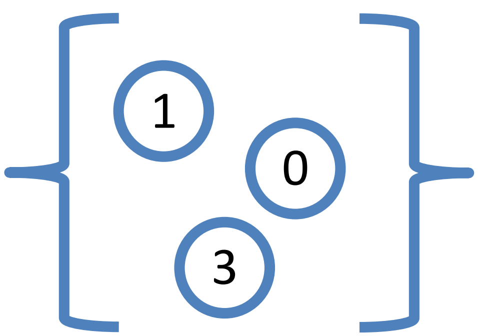
Abstract vs. Concrete ULSet

```
module ULSet : SET = struct
  type 'a set = 'a list
  let empty : 'a set = []
  let add (x:'a)  (s:'a set) :'a set =
     x::s (* can treat s as a list *)

end
```

```
s  =   0::3::1::[]
```

concrete representation

abstract view




```
module type SET = sig
  type 'a set
  val empty : 'a set
  val add   : 'a -> 'a set -> 'a set
end
```

```
(* A client of the ULSet module *)
;; open ULSet

let s : int set
   = add 0 (add 3 (add 1 empty))
```

Client code doesn't change!

Testing (and using) sets

- To use the values defined in the set module, use the “dot” syntax:

  `ULSet.`*<member>*

- Note: Module names must be capitalized in OCaml

```
let s1 = ULSet.add 3 ULSet.empty
let s2 = ULSet.add 4 ULSet.empty
let s3 = ULSet.add 4 s1

let test () : bool = (ULSet.member 3 s1)
;; run_test "ULSet.member 3 s1" test

let test () : bool = (ULSet.member 4 s3)
;; run_test "ULSet.member 4 s3" test
```

Testing (and using) sets

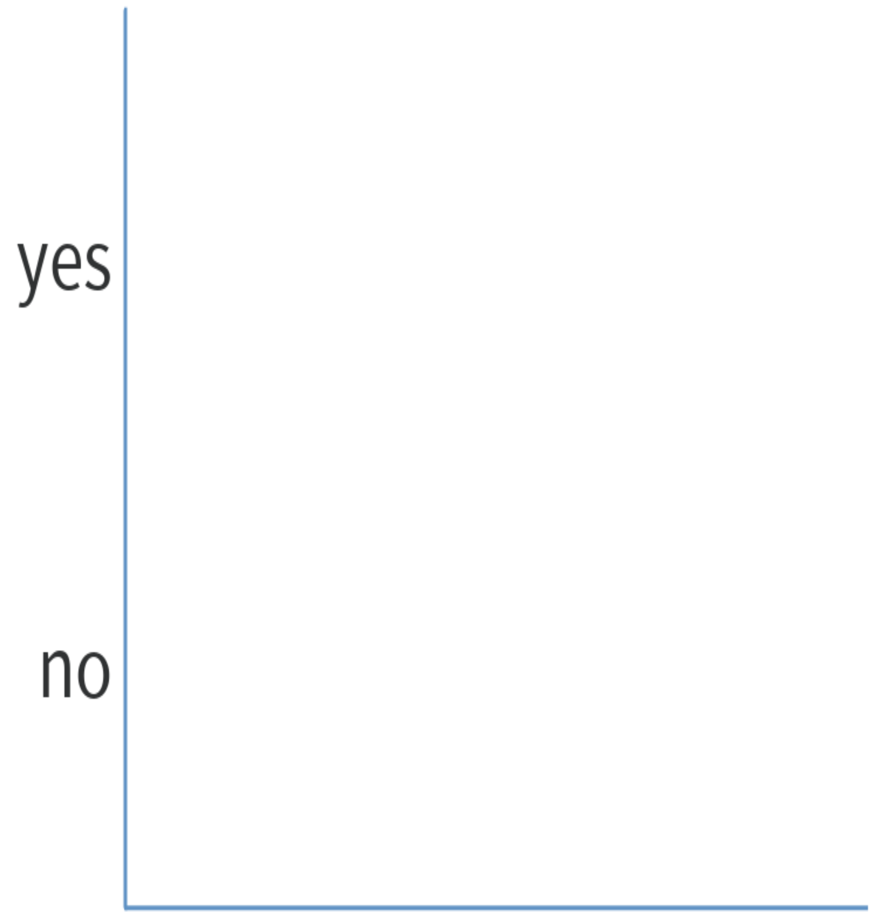
- Alternatively, use "open" to bring all of the names defined in the interface into scope.

```
;; open ULSet

let s1 = add 3 empty
let s2 = add 4 empty
let s3 = add 4 s1

let test () : bool = (member 3 s1)
;; run_test "ULSet.member 3 s1" test

let test () : bool = (member 4 s3)
;; run_test "ULSet.member 4 s3" test
```

Does this code typecheck?

```
module type SET = sig
  type 'a set
  val empty : 'a set
  val add   : 'a -> 'a set -> 'a set
end

module BSTSet : SET = struct
  type 'a tree =
    | Empty
    | Node of 'a tree * 'a * 'a tree
  type 'a set = 'a tree
  let empty : 'a set = Empty
  …
end
```

```
;; open BSTSet
let s1 : int set = add 1 empty
```



```
module type SET = sig
  type 'a set
  val empty : 'a set
  val add   : 'a -> 'a set -> 'a set
end

module BSTSet : SET = struct
  type 'a tree =
    | Empty
    | Node of 'a tree * 'a * 'a tree
  type 'a set = 'a tree
  let empty : 'a set = Empty
  …
end
```

Does this code type check?

```
;; open BSTSet
let s1 : int set = add 1 empty
```

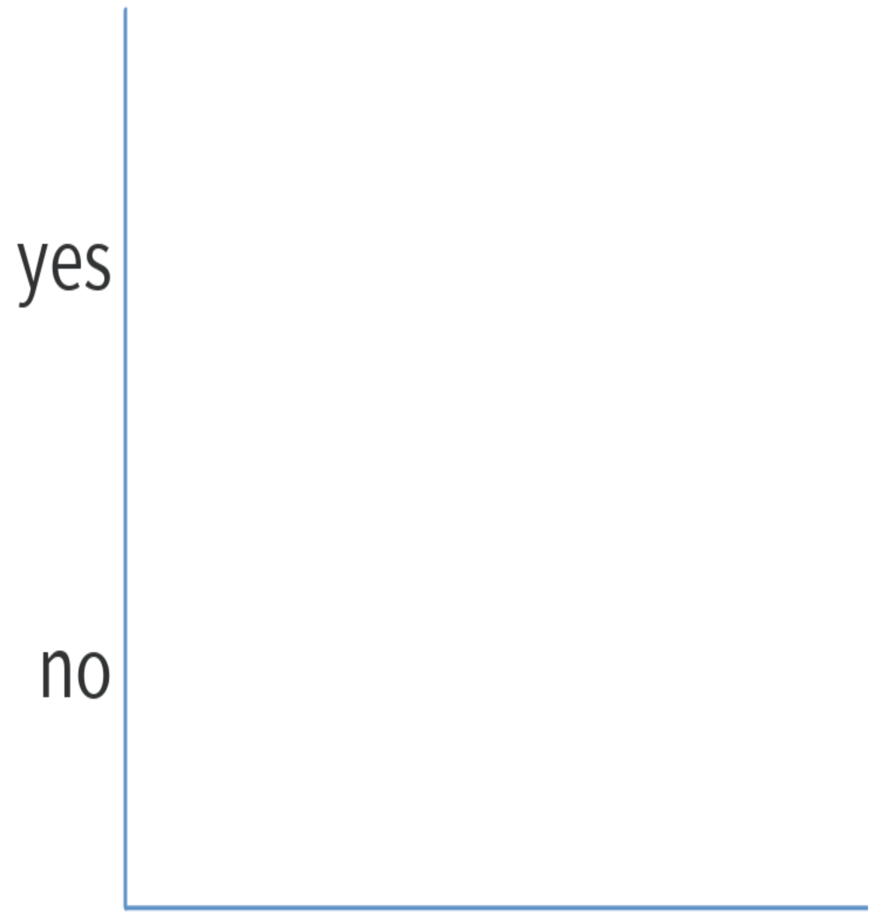
1. yes
2. no

Answer: yes

Does this code typecheck?

```
module type SET = sig
  type 'a set
  val empty : 'a set
  val add   : 'a -> 'a set -> 'a set
end

module BSTSet : SET = struct
  type 'a tree =
    | Empty
    | Node of 'a tree * 'a * 'a tree
  type 'a set = 'a tree
  let empty : 'a set = Empty
  …
end
```

```
;; open BSTSet
let s1 = add 1 empty
let i1 = begin match s1 with
         | Node (_,k,_) -> k
         | Empty -> failwith "impossible"
         end
```



```
module type SET = sig
  type 'a set
  val empty : 'a set
  val add   : 'a -> 'a set -> 'a set
end

module BSTSet : SET = struct
  type 'a tree =
    | Empty
    | Node of 'a tree * 'a * 'a tree
  type 'a set = 'a tree
  let empty : 'a set = Empty
  …
end
```

Does this code type check?

```
;; open BSTSet
let s1 = add 1 empty
let i1 = begin match s1 with
         | Node (_,k,_) -> k
         | Empty -> failwith "impossible"
         end
```

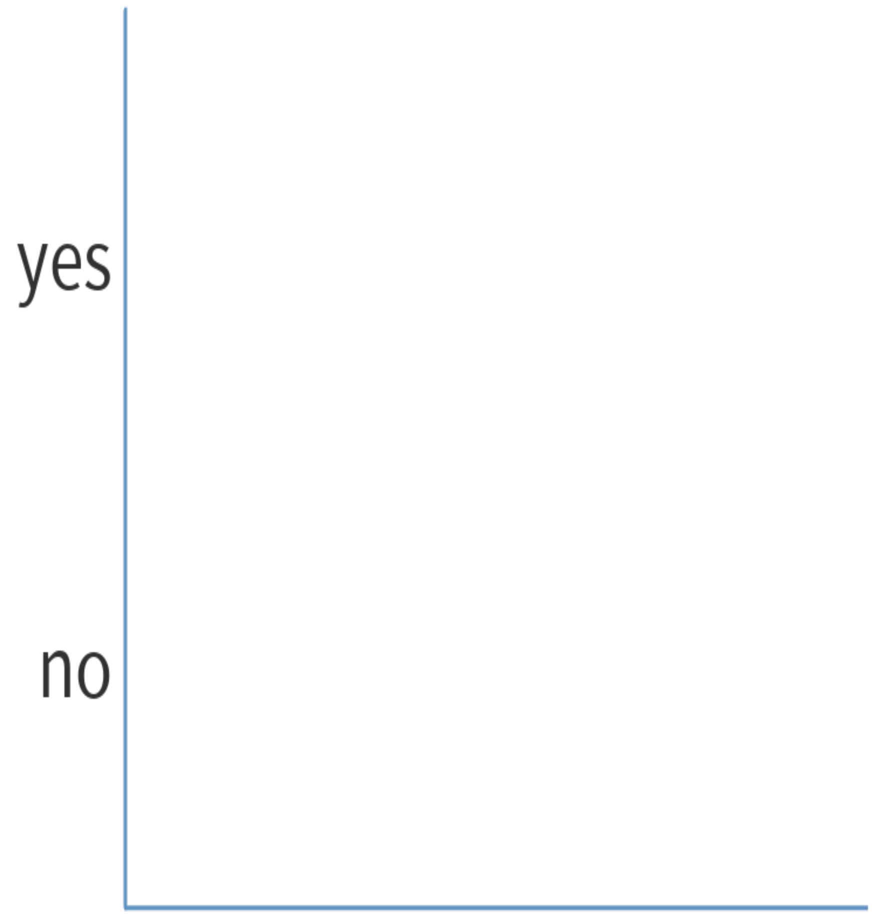
1. yes
2. no

Answer: no, add constructs a set, not a tree

Does this code typecheck?

```
module type SET = sig
  type 'a set
  val empty : 'a set
  val add   : 'a -> 'a set -> 'a set
end

module BSTSet : SET = struct
  type 'a tree =
    | Empty
    | Node of 'a tree * 'a * 'a tree
  type 'a set = 'a tree
  let empty : 'a set = Empty
  let size (t : 'a tree) : int = …
  …
end
```

```
;; open BSTSet
let s1 = add 1 empty
let i1 = size s1
```



```
module type SET = sig
  type 'a set
  val empty : 'a set
  val add   : 'a -> 'a set -> 'a set
end

module BSTSet : SET = struct
  type 'a tree =
    | Empty
    | Node of 'a tree * 'a * 'a tree
  type 'a set = 'a tree
  let empty : 'a set = Empty
  let size (t : 'a tree) : int = …
  …
end
```

Does this code type check?

```
;; open BSTSet
let s1 = add 1 empty
let i1 = size s1
```

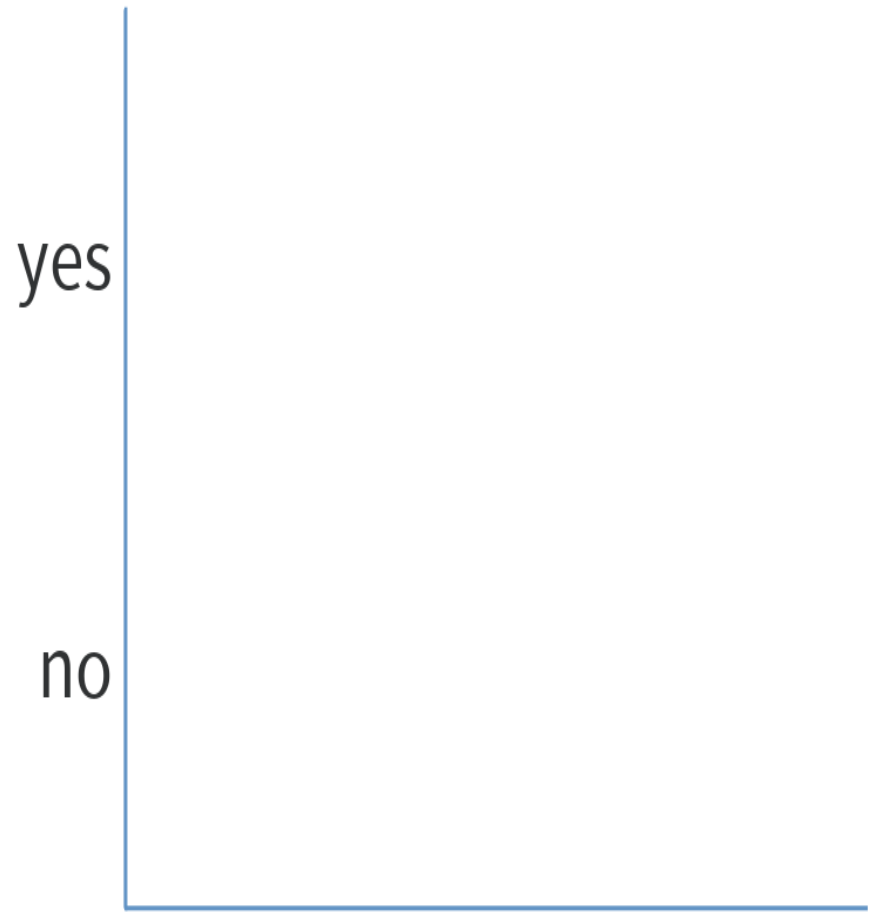
1. yes
2. no

Answer: no, cannot access helper functions outside the module

Does this code typecheck?

```
module type SET = sig
  type 'a set
  val empty : 'a set
  val add   : 'a -> 'a set -> 'a set
end

module BSTSet : SET = struct
  type 'a tree =
    | Empty
    | Node of 'a tree * 'a * 'a tree
  type 'a set = 'a tree
  let empty : 'a set = Empty
  …
end
```

```
;; open BSTSet
let s1 : int set = Empty
```



```
module type SET = sig
  type 'a set
  val empty : 'a set
  val add   : 'a -> 'a set -> 'a set
end

module BSTSet : SET = struct
  type 'a tree =
    | Empty
    | Node of 'a tree * 'a * 'a tree
  type 'a set = 'a tree
  let empty : 'a set = Empty
  …
end
```

Does this code type check?

```
;; open BSTSet
let s1 : int set = Empty
```

1. yes
2. no

Answer: no, the Empty data constructor is not available outside the module

If a client module works correctly and starts with:

```
;; open ULSet
```

will it continue to work if we change that line to:

```
;; open BSTSet
```

assuming that ULSet and BSTSet both implement SET and satisfy all of the set properties?

1. yes
2. no

Answer: yes (though performance may be different)

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

module BSTSet : SET = struct
  type 'a tree =
    | Empty
    | Node of 'a tree * 'a * 'a tree
  type 'a set = 'a tree
  let empty : 'a set = Empty
  …
end
```

Is is possible for a client to call **member** with a tree that is not a BST?

1. yes
2. no

No: the BSTSet operations preserve the BST invariants. there is no way to construct a non-BST tree using the interface.

Completing ULSet

See sets.ml

Abstract types

BIG IDEA: Hide the *concrete representation* of a type behind an *abstract interface* to preserve invariants

- The interface **restricts** how other parts of the program can interact with the data
  - Type checking ensures that the **only** way to create a set is with the operations in the interface
  - If all operations preserve invariants, then all sets in the program must satisfy invariants
  - Example: all BST-implemented sets must satisfy the BST invariant, therefore the lookup function can assume that its input satisfies the invariant
- Benefits:
  - **Safety**: The other parts of the program can't cause bugs in the set implementation
  - **Modularity**: It is possible to change the implementation without changing the rest of the program

Summary: Abstract Types

- Different programming languages have different ways of letting you define abstract types
- At a minimum, this means providing:
  - A way to specify (write down) an interface
  - A means of hiding implementation details (*encapsulation*)
- In OCaml:
  - Interfaces are specified using a *signature* or *interface*
  - Encapsulation is achieved because the interface can *omit* information
    - type definitions
    - names and types of auxiliary functions
  - Clients *cannot* mention values or types not named in the interface

Bonus Material: OCaml Details

module and interface files

.ml and .mli files

- You've already been using signatures and modules in OCaml.

- A series of type and `val` declarations stored in a file `foo.mli` is considered as defining a signature FOO
- A series of top-level definitions stored in a file `foo.ml` is considered as defining a module Foo

foo.mli

```ocaml
type t
val z : t
val f : t -> int
```

foo.ml

```ocaml
type t = int
let z : t = 0
let f (x:t) : int =
  x + 1
```

test.ml

```ocaml
;; open Foo
;; print_int
      (Foo.f Foo.z)
```

Files

```ocaml
module type FOO = sig
  type t
  val z : t
  val f : t -> int
end

module Foo : FOO = struct
  type t = int
  let z : t = 0
  let f (x:t) : int =
    x + 1
end

module Test = struct
  ;; open Foo
  ;; print_int
        (Foo.f Foo.z)
end
```