Programming Languages and Techniques (CIS120)

Lecture 11

Review: Abstract types
Finite Maps

Midterm 1

- Friday, September 27^{th}
- Coverage: up to Monday, Sept. 23 (Chs. 1-10)
- Time: During lecture (001 @ 11am, 002 @ noon)
  Last names: A – L Leidy Labs 10
  Last names: M – Z Stitler (STIT) B6
- Review Session: Wednesday 6:00-8:00pm Towne 100
- Review Material:
  – old exams on the web site lecture schedule
- Makeup exam
  – Monday, Sept. 30^{th}
  – sign up form on the web site

Announcements

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

- Homework 4
  - Available soon after exam
  - Due: Tuesday, Oct. 8th

Review: Abstract types (e.g. set)

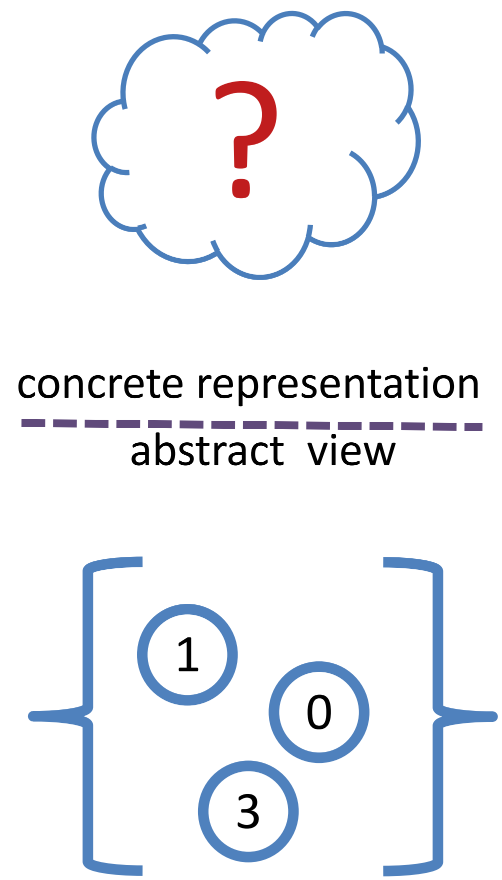
- An abstract type is defined by its *interface* and its *properties*, not its representation.

- Interface: defines operations on the type
  - 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 (possibly with invariants) that satisfies the interface and properties can be a set.

- *Clients of an implementation can only access what is explicitly in the abstract type's interface*



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

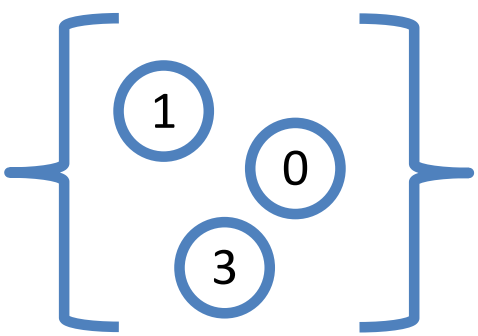
```
s  =   0::3::1::[]
```

```
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
```

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

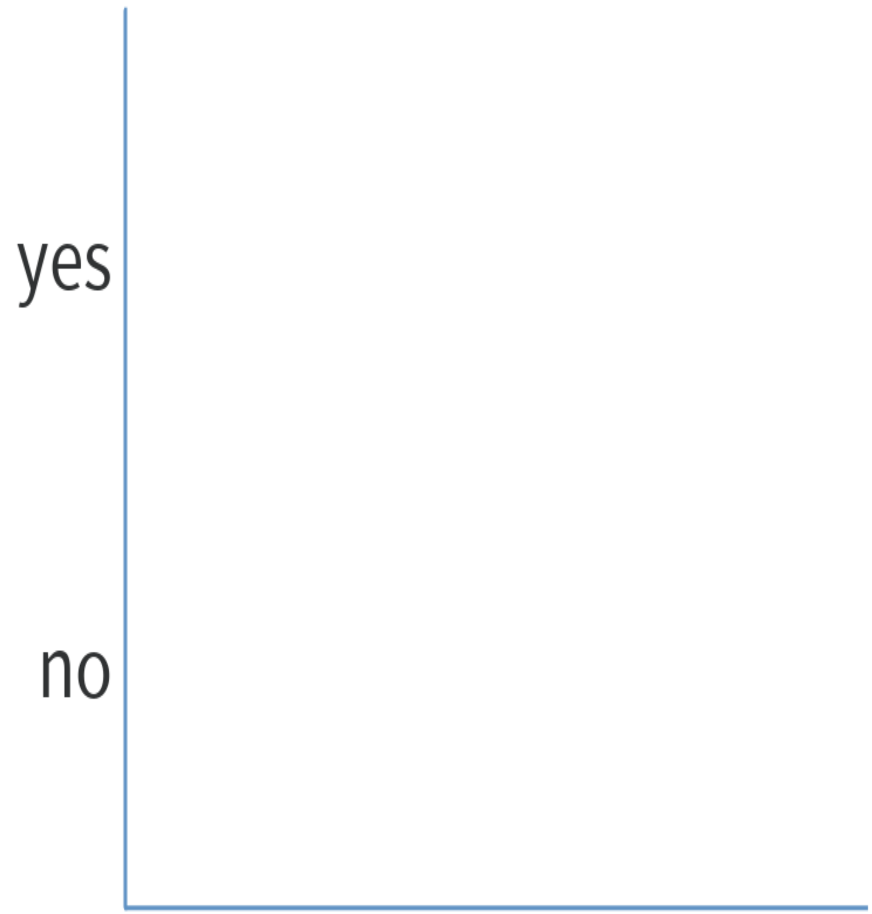
- Alternatively, use "open" to bring all of the names defined in the interface into scope. (Saves on repeating "ULSet.")

```
;; 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
```



Start the presentation to see live content. Still no live content? Install the app or get help at **PollEv.com/app**

Total Results

```
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
```

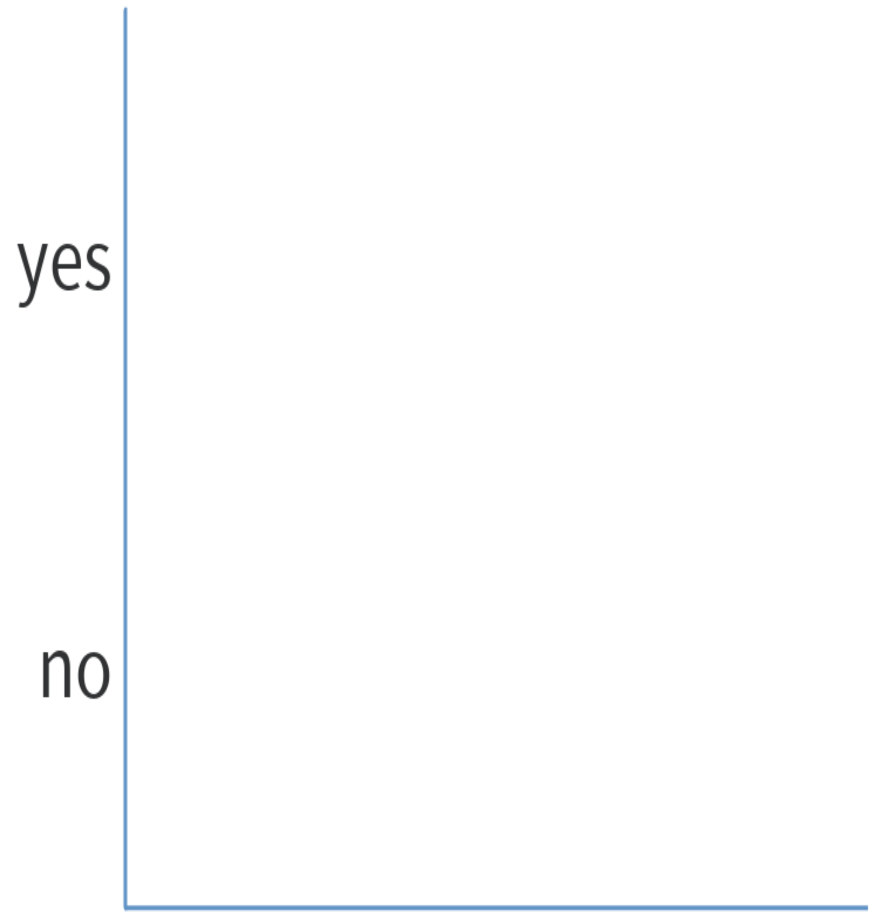
1. yes
2. no

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
```



Start the presentation to see live content. Still no live content? Install the app or get help at **PollEv.com/app**

Total Results

```
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
```

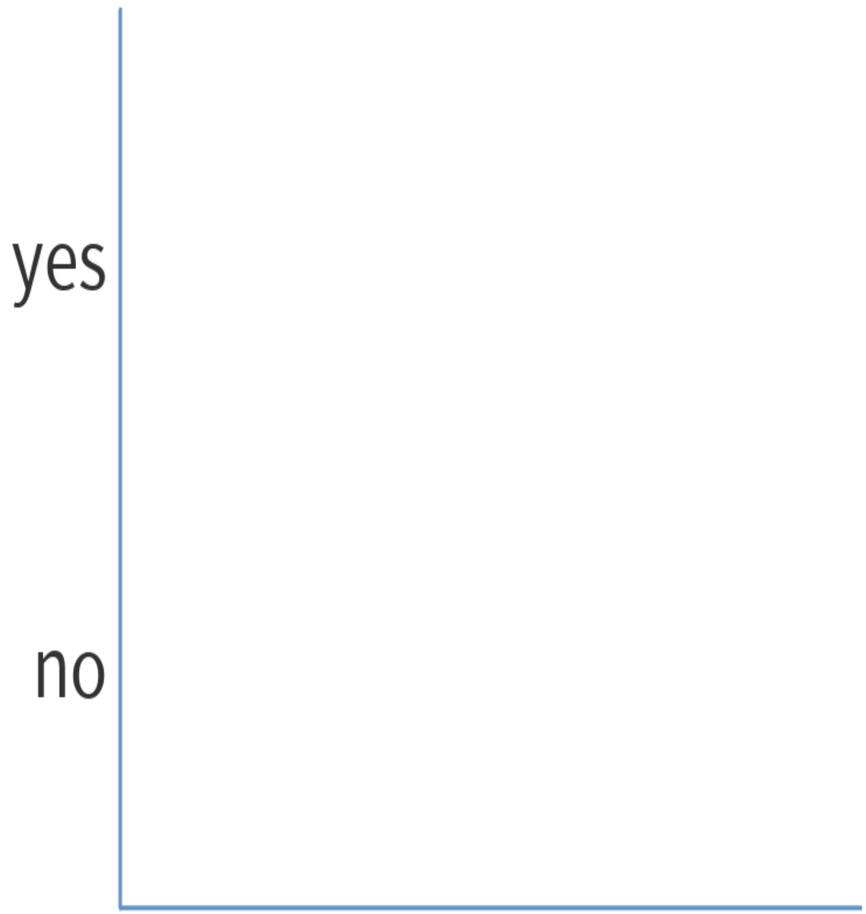
1. yes
2. no

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
```



Start the presentation to see live content. Still no live content? Install the app or get help at **PollEv.com/app**

Total Results

```
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
```

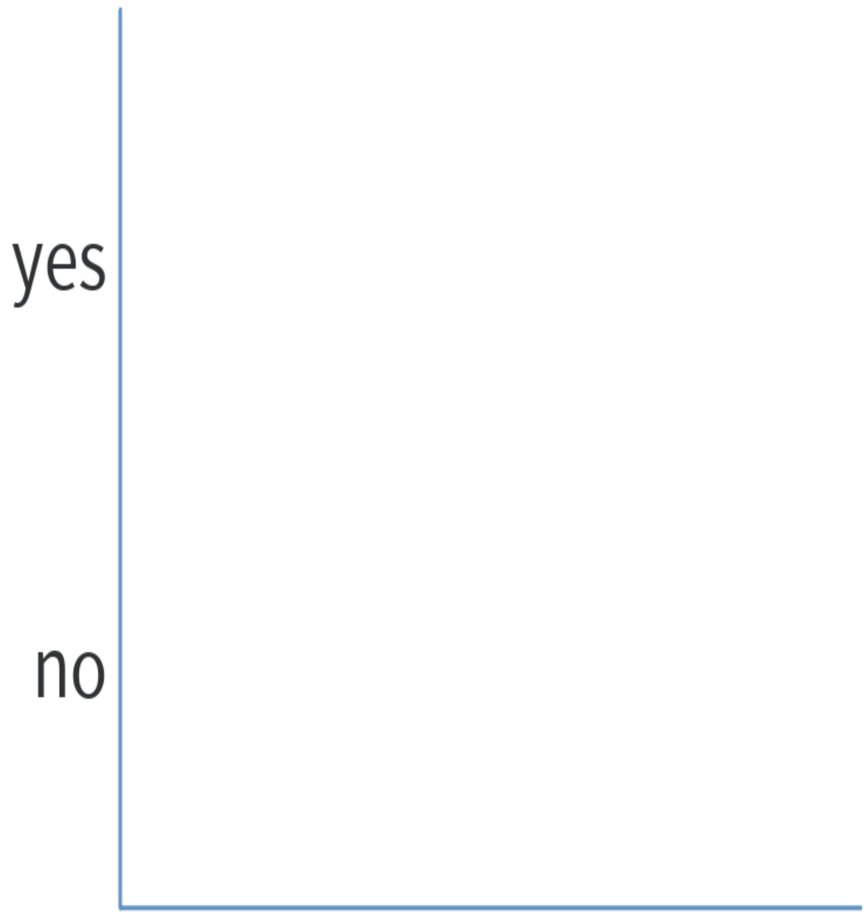
1. yes
2. no

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.

What Should You Test?

- Interface: defines operations on the type
- Properties: define how the operations interact
  - 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

Test the properties!

A *property* is a general statement about the behavior of the interface: For *any* set `s` and *any* element `x`:

```
member x (add x s) = true
```

A (good) test case checks a specific instance of the property:

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

Property-based Testing

1. Translate informal requirements into general statements about the interface.

> Example: "Order doesn't matter" becomes
> For *any* set `s` and *any* elements `x` and `y`,
> `add x (add y s)` equals `add y (add x s)`

2. Write tests for the "interesting" instances of the general statement.

> Example. "interesting" choices:
> `s` = `empty`, `s` = nonempty,
> `x = y`, `x <> y`
> one or both of `x`, `y` already in `s`

Notes:

- one can't (usually) exhaustively test all possibilities (too many!) so instead, cover the "interesting" possibilities
- be careful with equality! `ULSet.equal` is *not* the same as =.

Completing ULSet

See sets.ml

Finite Maps

Another example of abstract datatype interfaces
& concrete implementations

Motivating Scenario

- Suppose you were writing some course-management software and needed to look up the lab section for a student given the student's PennKey?
  - Students might add/drop the course
  - Students might switch lab sections
  - Students should be in only *one* lab section

- How would you do it? What data structure would you use?

Example

Key	Value
“stephanie”	15
“mitch”	05
“ezaan”	10
“likat”	15
…	…

- Each key is associated with a value.
  - No two keys are identical
  - Values can be repeated
- Given the key “stephanie" we want to find / lookup the value 15

Finite Maps

- A *finite map* (a.k.a. *dictionary*) is a collection of *bindings* from distinct *keys* to *values*.
  - Operations to *add* & *remove* bindings, *test* for key membership, *look up* the value bound to a particular key
- Example: a `(string, int) map` might map a PennKey to the lab section.
  - The map type is generic in *two* arguments
- Like sets, finite maps appear in many settings to map:
  - domain names to IP addresses
  - words to their definitions (a dictionary)
  - user names to passwords
  - game character unique identifiers to dialog trees
  - …

Signature: Finite Map

```
module type MAP = sig

  type ('k,'v) map

  val empty   : ('k,'v) map
  val add     : 'k -> 'v -> ('k,'v) map -> ('k,'v) map
  val remove  : 'k       -> ('k,'v) map -> ('k,'v) map
  val mem     : 'k -> ('k,'v) map -> bool
  val get     : 'k -> ('k,'v) map -> 'v
  val entries : ('k,'v) map -> ('k * 'v) list
  val equals  : ('k,'v) map -> ('k,'v) map -> bool

end
```

Properties of Finite Maps

For any finite map m, key k, and value v:

1. `get k (add k v m) = v`
2. If `k1 <> k2` then
   `get k1 (add k2 v2 (add k1 v1 m)) = v1`
3. if `mem k m = true` then
   there is a `v` such that `get k m = v`
4. If `mem k m = false` then
   `get k m = v` fails
5. `mem k (add k v m) = true`
6. `mem k (remove k m) = false`

And others…

Tests for Finite Map abstract type

```
;; open Assert

(* Specifying the properties of the MAP abstract type via test cases. *)

(* A simple map with one element. *)
let m1 : (int,string) map = add 1 "uno" empty

(* list entries for this simple map *)
;; run_test "entries m1" (fun () -> entries m1 = [(1,"uno")])

(* access value for key in the map *)
;; run_test "find 1 m1" (fun () -> (get 1 m1) = "uno")

(* find for value that does not exist in the map? *)
;; run_failing_test "find 2 m1" (fun () -> (get 2 m1) = "dos" )

let m2 : (int, string) map = add 1 "un" m1

(* find after redefining value, should be new value *)
;; run_test "find 1 m2" (fun () -> (get 1 m2) = "un")

(* entries after redefining value, should only show new value *)
;; run_test "entries m2" (fun () -> entries m2 = [(1, "un")])
```

Implementation: Ordered Lists

```
module Assoc : MAP = struct

  (* Represent a finite map as a list of pairs. *)
  (* Representation invariant:                  *)
  (*   - no duplicate keys (helps get, remove)  *)
  (*   - keys are sorted (helps equals, helps get) *)

type ('k,'v) map = ('k * 'v) list

  let empty : ('k,'v) map = []

  let rec mem (key:'k) (m : ('k,'v) map) : bool =
    begin match m with
    | [] -> false
    | (k,v)::rest ->
      (key >= k) &&
         ((key = k) || (mem key rest))
    end

;; run_test "mem test" (fun () -> mem "b" [("a",3); ("b",4)])
```

Implementation: Ordered Lists

```
let rec get (key:'k) (m : ('k,'v) map) : 'v =
  begin match m with
  | [] -> failwith "key not found"
  | (k,v)::rest ->
    if key < k then failwith "key not found"
    else if key = k then v
    else get key rest
  end

let rec remove (key:'k) (m : ('k,'v) map) : ('k,'v) map =
  begin match m with
  | [] -> []
  | (k,v)::rest ->
    if key < k then m
    else if key = k then rest
    else (k,v)::remove key rest
  end
```

Completing module implementation

finiteMap.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

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

foo.ml

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

test.ml

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

Files

```
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
```