Programming Languages and Techniques (CIS1200)

Lecture 9

Higher-order functions: transform and fold

Lecture notes: Chapter 9

Announcements (1)

- Please complete the intro survey (link on Ed) available this afternoon
- Homework 3 available this afternoon
  - Practice with BSTs, generic functions, first-class functions, and abstract types
  - Due Tuesday, September 24th at 11:59pm
  - *Start early!*
    - *Problems 1-4 can be done after class today*
    - *Problems 5-8 can be done after class on Friday*
- Reading: Chapters 8, 9, and 10 of the lecture notes

Announcements (2)

- Midterm 1: Friday, September 27th
  - Coverage: up to Wednesday, Sep 25th (Chapters 1-10)
  - During lecture
    Last names: A – Z Meyerson Hall B1
  - 60 minutes; closed book, closed notes
  - Review Material
    - old exams on the web site ("schedule" tab)
  - Review Session
    - Wednesday, September 25, 7:00-9:00pm, Towne 100 (will be recorded)
    - Review Videos will be posted this weekend

First-Class Functions

First-class Functions

function body

```
let f : t -> u = fun (x:t) -> <body>
```

function type

anonymous function value

- Functions are *first-class values* in OCaml: they can be manipulated like any other value.
- They have a type that specifies the input and output types.
- The “fun” keyword introduces an *anonymous function*.
  - Sometimes called *lambdas** or *closures*

*The term “lambda” comes from Church’s *lambda calculus*.

2 = 1 + 1

A function that takes *two* arguments…

```
int -> int -> int
```

has the same type as a function that takes *one* argument and returns a function that takes *one* argument

```
int -> (int -> int)
```

This is actually useful!

Multiple Arguments

We can decompose a standard function definition

```
let sum (x : int) (y:int) : int = x + y
```

into parts:

```
let sum = fun (x:int) -> fun (y:int) -> x + y
```

define a variable with that value

create a function value

that returns a function value

The two definitions of sum have the same type and behave the same!

```
let sum : int -> int -> int
```

Partial Application

```
let sum (x : int) (y:int) : int = x + y
```

```
sum 3
⟼ (fun (x:int) -> fun (y:int) -> x + y) 3      definition
⟼  fun (y:int) -> 3 + y                  substitute 3 for x
```

the result of a "partially applied function" is itself a function (that can later be applied)

Functions that return functions

```
let sum (x : int) (y:int) : int = x + y
```

```
let sum = fun (x:int) -> fun (y:int) -> x + y
```

```
sum 3
⟼ (fun (x:int) -> fun (y:int) -> x + y) 3
⟼ fun (y:int) -> 3 + y
```

definition

substitute 3 for x

the result of a "partially applied function" is itself a function (that can later be applied)

List transformations

A fundamental design pattern
using first-class functions

Phone book example

```
type entry = string * int
let phone_book = [ ("Steve", 2155559092), … ]

let rec get_names (p : entry list) : string list =
  begin match p with
  | ((name, num)::rest) -> name :: get_names rest
  | [] -> []
  end

let rec get_numbers (p : entry list) : int list =
  begin match p with
  | ((name, num)::rest) -> num :: get_numbers rest
  | [] -> []
  end
```

Can we use first-class functions to refactor code to share common structure?

Refactoring

```
let rec helper (f:entry -> 'b) (p:entry list) : 'b list =
  begin match p with
  | (entry::rest) -> f entry :: helper f rest
  | [] -> []
  end

let get_names (p : entry list) : string list =
  helper fst p
let get_numbers (p : entry list) : int list =
  helper snd p
```

fst and snd are functions that access the parts of a tuple:

```
let fst (x,y) = x
let snd (x,y) = y
```

The argument `f` determines what happens with the entry at the head of the list

Going even more generic

```
let rec helper (f:entry -> 'b) (p:entry list) : 'b list =
  begin match p with
  | (entry::rest) -> f entry :: helper f rest
  | [] -> []
  end

let get_names (p : entry list) : string list =
  helper fst p
let get_numbers (p : entry list) : int list =
  helper snd p
```

Now let's make it work for *all* lists, not just lists of entries…

Going even more generic

```
let rec helper (f:'a -> 'b) (p:'a list) : ’b list =
  begin match p with
  | (entry::rest) -> f entry :: helper f rest
  | [] -> []
  end

let get_names (p : entry list) : string list =
  helper fst p
let get_numbers (p : entry list) : int list =
  helper snd p
```

'a stands for (string*int)
'b stands for int

snd : (string*int) -> int

Transforming Lists

```
let rec transform (f: 'a->'b) (l:'a list) : 'b list =
  begin match l with
  | []   -> []
  | h::t -> (f h)::(transform f t)
  end
```

List *transformation*

a.k.a. "*mapping** a function across a list"

- foundational function for programming with lists
- part of OCaml standard library (called List.map)
- used over and over again

(e.g., Google's famous *map*-reduce infrastructure)

*many languages (including OCaml) use the terminology "map" for the function that transforms a list by applying a function to each element. Don't confuse List.map with "finite map".

9: What is the value of this expresssion?

✓ 0

[0; -1; 1; -2] 0%

[1] 0%

[1; 1; 0; 1] 0%

[false; false; true; false] 0%

runtime error 0%

What is the value of this expresssion?

```
transform (fun (x:int) -> x > 0)
    [0 ; -1; 1; -2]
```

1. [0; -1; 1; -2]
2. [1]
3. [1; 1; 0; 1]
4. [false; false; true; false]
5. runtime error

ANSWER: 4

The ‘`fold`’ design pattern

a general-purpose recursive function

Refactoring code, again

Is there a pattern in the definition of these two functions?

```
let rec exists (l : bool list) : bool =
   begin match l with
   | [] -> false
   | h :: t -> h || exists t
   end
```

```
let rec acid_length (l : acid list) : int =
   begin match l with
   | [] -> 0
   | h :: t -> 1 + acid_length t
   end
```

Refactoring code, again

Is there a pattern in the definition of these two functions?

```
let rec exists (l : bool list) : bool =
   begin match l with
   | [] -> false
   | h :: t -> h || exists t
   end
```

```
let rec acid_length (l : acid list) : int =
   begin match l with
   | [] -> 0
   | h :: t -> 1 + acid_length t
   end
```

base case:
Simple answer when the list is empty

combine step:
Do something with the head of the list and the result of the recursive call

Can we factor out this pattern using first-class functions?

Preparation

```
let rec exists (l : bool list) : bool =
   begin match l with
   | [] -> false
   | h :: t -> h || exists t
   end
```

```
let rec acid_length (l : acid list) : int =
   begin match l with
   | [] -> 0
   | h :: t -> 1 + acid_length t
   end
```

Preparation: introduce a helper

```
let rec helper (l : bool list) : bool =
   begin match l with
   | [] -> false
   | h :: t -> h || helper t
   end

let exists (l : bool list) = helper l
```

First: introduce a helper function that will (eventually) become the same for both definitions.

```
let rec helper (l : acid list) : int =
   begin match l with
   | [] -> 0
   | h :: t -> 1 + helper t
   end

let acid_length (l : acid list) = helper l
```

Abstracting with respect to Base

```
let rec helper (l : bool list) : bool =
   begin match l with
   | [] -> false
   | h :: t -> h || helper t
   end

let exists (l : bool list) = helper l
```

```
let rec helper (l : acid list) : int =
   begin match l with
   | [] -> 0
   | h :: t -> 1 + helper t
   end

let acid_length (l : acid list) = helper l
```

Abstracting with respect to Base

```
let rec helper (base : bool) (l : bool list) : bool =
   begin match l with
   | [] -> base
   | h :: t -> h || helper base t
   end

let exists (l : bool list) = helper false l
```

```
let rec helper (base : int) (l : acid list) : int =
   begin match l with
   | [] -> base
   | h :: t -> 1 + helper base t
   end

let acid_length (l : acid list) = helper 0 l
```

Abstracting with respect to Combine

```
let rec helper (base : bool) (l : bool list) : bool =
   begin match l with
   | [] -> base
   | h :: t -> h || helper base t
   end

let exists (l : bool list) = helper false l
```

```
let rec helper (base : int) (l : acid list) : int =
   begin match l with
   | [] -> base
   | h :: t -> 1 + helper base t
   end

let acid_length (l : acid list) = helper 0 l
```

Abstracting with respect to Combine

```
let rec helper (base : bool) (l : bool list) : bool =
   begin match l with
   | [] -> base
   | h :: t -> h || helper base t
   end

let exists (l : bool list) = helper false l
```

```
let rec helper (base : int) (l : acid list) : int =
   begin match l with
   | [] -> base
   | h :: t -> 1 + helper base t
   end

let acid_length (l : acid list) = helper 0 l
```

Abstracting with respect to Combine

```
let rec helper (combine : bool -> bool -> bool)
               (base : bool) (l : bool list) : bool =
   begin match l with
   | [] -> base
   | h :: t -> combine h (helper combine base t)
   end

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

```
let rec helper (combine : acid -> int -> int)
               (base : int) (l : acid list) : int =
   begin match l with
   | [] -> base
   | h :: t -> combine h (helper combine base t)
   end

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

What about the types?

```
let rec helper (combine : bool -> bool -> bool)
               (base : bool) (l : bool list) : bool =
   begin match l with
   | [] -> base
   | h :: t -> combine h (helper combine base t)
   end

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

```
let rec helper (combine : acid -> int -> int)
               (base : int) (l : acid list) : int =
   begin match l with
   | [] -> base
   | h :: t -> combine h (helper combine base t)
   end

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

What about the types?

```
let rec helper (combine : bool -> bool -> bool)
               (base : bool) (l : bool list) : bool =
   begin match l with
   | [] -> base
   | h :: t -> combine h (helper combine base t)
   end

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

```
let rec helper (combine : acid -> int -> int)
               (base : int) (l : acid list) : int =
   begin match l with
   | [] -> base
   | h :: t -> combine h (helper combine base t)
   end

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

Making the Helper Generic

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

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

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

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

The helpers now have the *same* type.

But they are instantiated differently for the two uses.

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
```

Just rename "helper" to "fold".

fold (a.k.a. "reduce")

- Like `transform`, foundational function for programming with lists
- Captures the pattern of *recursion over lists*
- Part of OCaml standard library (`List.fold_right`)
- Similar operations for other recursive datatypes (`fold_tree`)

Using List Fold

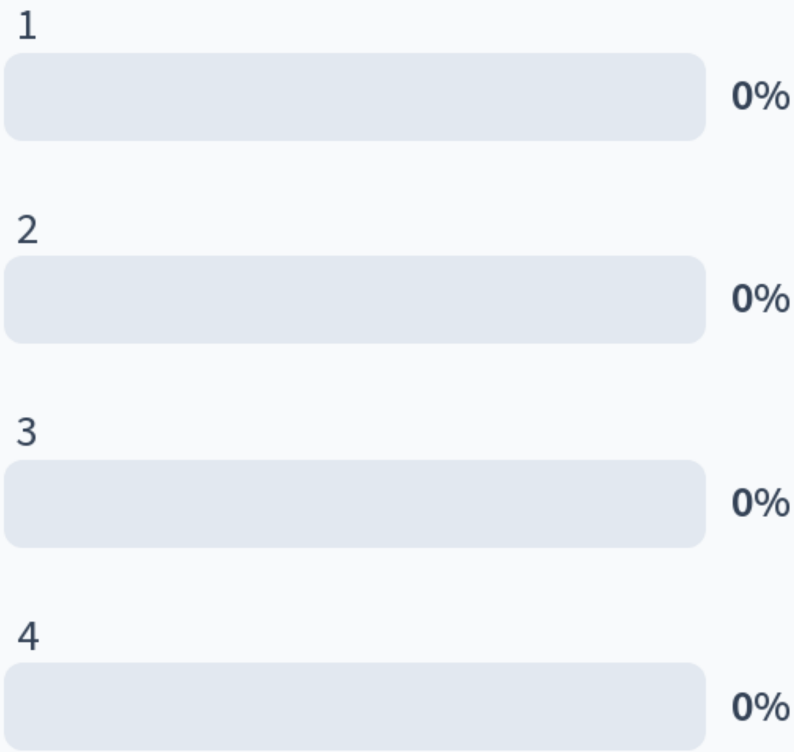
```
let rec fold (combine: 'a -> 'b -> 'b)
             (base:'b) (l : 'a list) : 'b =
  begin match l with
  | [] -> base
  | x :: t -> combine x (fold combine base t)
  end

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

fold:
general-purpose recursion

combine function:
computes the result given h the head of the list and acc the "*accumulated*" answer given by recursion

base case:
answer when the list is empty

9: 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`.

Answer: 2

9: Rewrite using fold

0

1	0%
2	0%
3	0%
4	0%

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

Functions as Values

- We've seen many ways in which functions can be treated as values in OCaml
- Everyday programming practice (in many languages, not just OCaml!) 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 for large-scale computing: 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

Chapter 10

You are probably familiar with the idea of a *set* from mathematics.

In math, we typically write sets like this:

$\emptyset$ {1,2,3,4} {true,false} {X,Y,Z}

operations:

$S \cup T$ for *union* and

$S \cap T$ for *intersection*;

we write $x \in S$ for the predicate

"x is a member of the set S"

A *set* is an abstraction

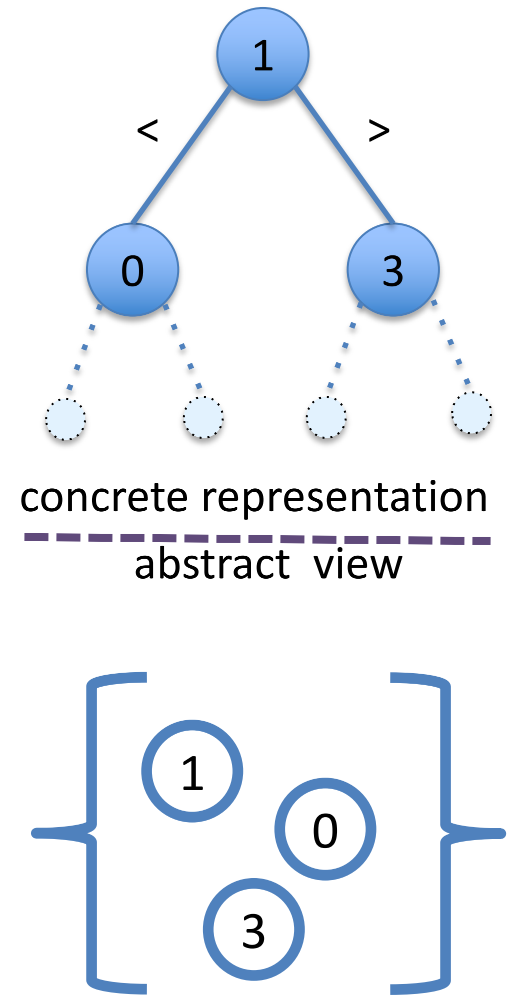
- A set is a collection of data
  - we have operations for forming sets of elements
  - we can ask whether elements are in a set
- A set is a lot like a list, except:
  - Order doesn't matter
  - Duplicates don't matter
  - *It isn't built into OCaml*
- Sets show up frequently in applications
  - Examples: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

An element's *presence* or *absence* in the set is all that matters…

Abstract type: set

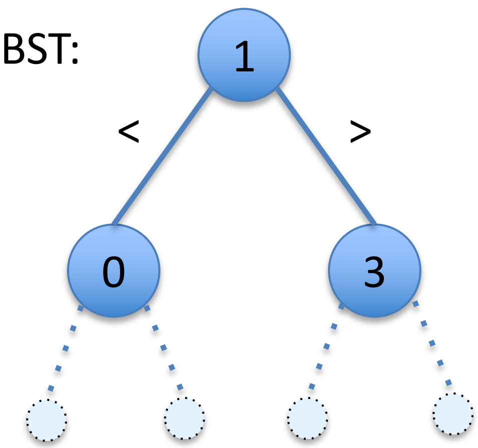
- A BST can *implement (represent)* a *set*
  - *there is a way to represent an empty set (Empty)*
  - *there is a way to list all elements contained in the set (inorder)*
  - *there is a way to test membership (lookup)*
  - *Can define union/intersection (with insert and delete)*

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



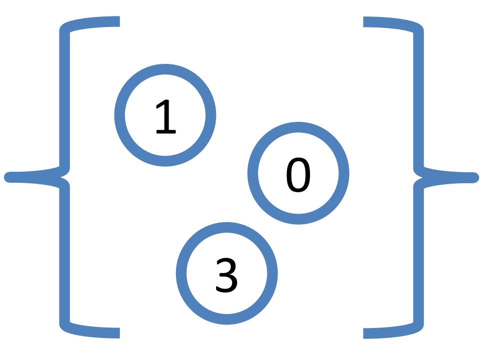
Three Representations of Sets

BST:



concrete representation

abstract view

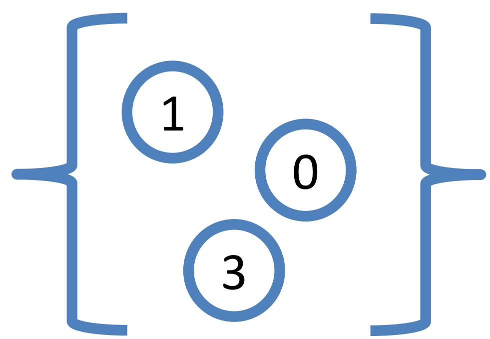


Alternate representation: unsorted linked list.

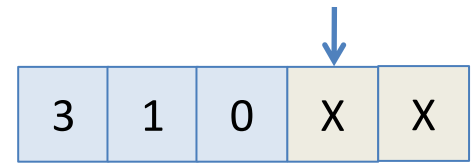
```
3::0::1::[]
```

concrete representation

abstract view

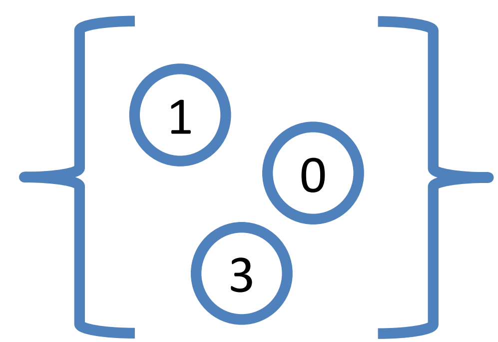


Alternate representation: reverse sorted array with Index of next slot.



concrete representation

abstract view

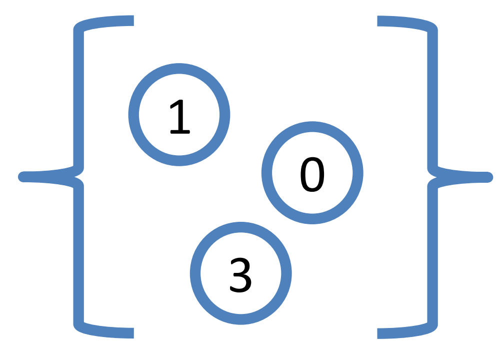


Abstract types (e.g. set)

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

concrete representation

abstract view



Sets in OCaml

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?*

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 all the operations *)
  …
end
```

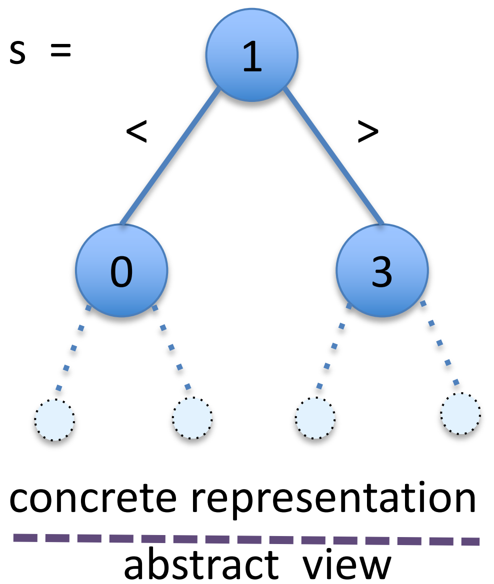
Implementing the set Module

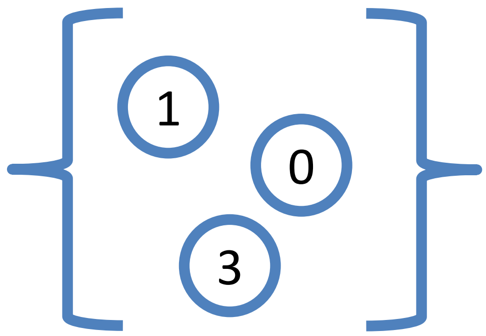
```
module BSTSet : SET = struct

  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 has to 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 interface

Abstract vs. Concrete BSTSet






```
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

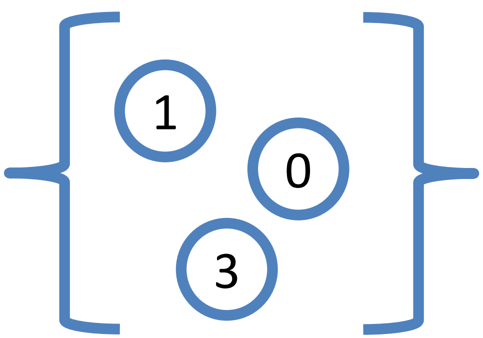
Abstract vs. Concrete ULSet

```
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!