

# Programming Languages and Techniques (CIS120)

## Lecture 10

February 6<sup>th</sup>, 2015

Abstract types: sets

Lecture notes: Chapter 10

# Announcements

- Homework 3 is available
  - due *Thursday, Sept. 24<sup>th</sup>* at 11:59:59pm
- Read Chapter 10 of lecture notes
- Midterm 1
  - Scheduled in class on *Friday, October 2<sup>nd</sup>*
  - Contact me if you need to take the make-up exam
  - More details to follow!

# 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`)

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 by with fold.

Answer: 2

How would you rewrite this function

```
let rec join (l:string list) : string =  
  begin match l with  
  | [] -> ""  
  | s :: [] -> s  
  | s1 :: rest -> s1 ^ "," ^ (join rest)  
  end
```

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

1. combine is: `(fun (s:string) (acc:string) -> s ^ "," ^ acc)`  
base is: `""`
2. combine is: `(fun (s1:string) (acc:string list) ->  
if acc = s::[] then s else s1 ^ "," ^ acc)`  
base is: `""`
3. combine is: `(fun (s:string) (acc:string) -> s)`  
base is: `[]`
4. join can't be written by with fold.

Answer: 4

# Functions as Data

- We've seen a number of ways in which functions can be treated as data in OCaml
- Present-day programming practice offers many more examples at the “small scale”:
  - objects bundle “functions” (a.k.a. methods) with data
  - iterators (“cursors” for walking over data structures)
  - event listeners (in GUIs)
  - etc.
- The idiom is useful at the “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

Are you familiar with the idea of a *set* from mathematics?

1. yes
2. no

In math, we typically write sets like this:

$\emptyset$   $\{1,2,3\}$   $\{\text{true},\text{false}\}$

with operations:

$S \cup T$  for union and

$S \cap T$  for intersection;

we write  $x \in S$  for

“ $x$  is a member of the set  $S$ ”

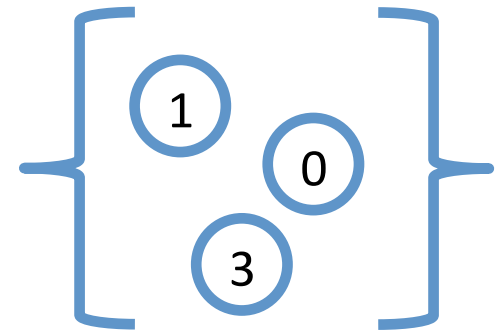
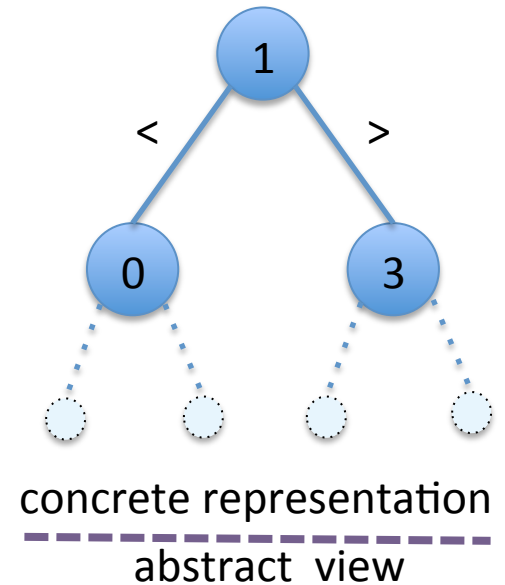
# A *set* is an abstraction

- 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*

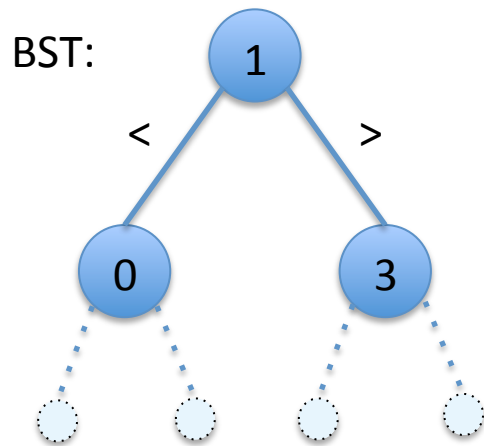
} An element's *presence* or *absence* in the set is all that matters...
- Sets show up frequently in applications
  - Examples: set of students in a class, set of coordinates in a graph, set of answers to a survey, set of data samples from an experiment, ...

# Abstract type: set

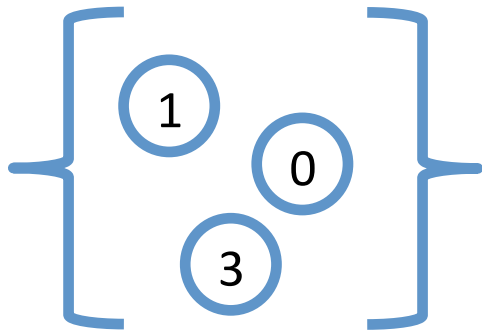
- A BST can *implement (represent)* a set
  - there is a way to represent an empty set (*Empty*)
  - there is a way to list all elements contained in the set (*inorder*)
  - there is a way to test membership (*lookup*)
  - could define union/intersection (*insert and delete*)
- Order doesn't matter
  - We create BSTs by adding elements to an empty BST
  - The BST data structure doesn't remember what order we added the elements
- Duplicates don't matter
  - Our implementation doesn't keep track of how many times an element is added
  - BST invariant ensure that each node is unique
- *BSTs are not the **only** way to implement sets*



# Three Example Representations of Sets



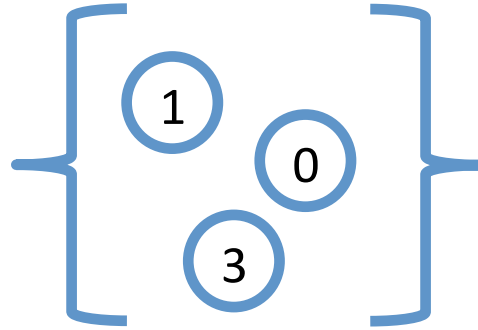
concrete representation  
-----  
abstract view



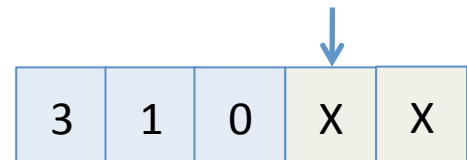
Alternate representation:  
unsorted linked list.

3::0::1::[]

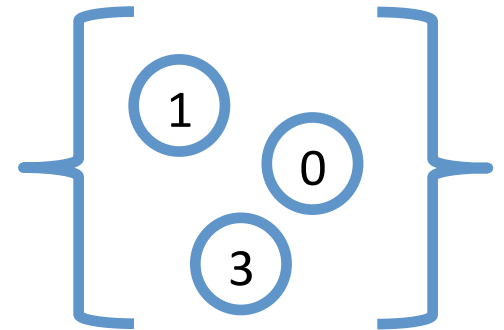
concrete representation  
-----  
abstract view



Alternate representation:  
reverse sorted array with  
index to next slot.



concrete representation  
-----  
abstract view

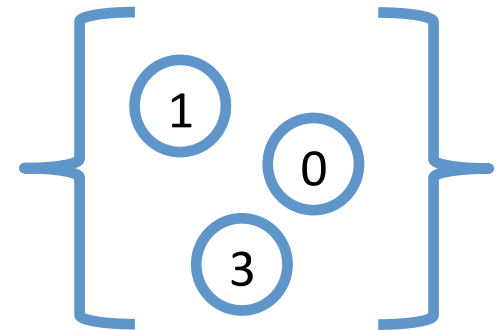


# Abstract types (e.g. set)

- 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 action

# A design problem

*As a high-school student, Stephanie had the job of reading books and finding which words, out of a collection of the 1000-most common SAT vocabulary words, appeared in a particular book. She enjoyed being paid to read, but she would have enjoyed being paid to program more. How could she have automated this task?*

1. What are the important concepts or *abstractions* for this problem?
  - The list of words that appear in a book
  - The set of 1000-most common SAT words
  - The set of words from the list that are contained in the set

## 2. Formalize the Interface

- Suppose we had a generic type of sets:

`'a set`

- We can formalize the interface for our problem:

```
let findVocab (text : string list)  
              (vocab : string set)  
              string set =  
  failwith "write me"
```

### 3. Write Test Cases

Test cases specify the *interface* and *properties* of the necessary abstractions.

```
let vocab : string set =  
  set_of_list ["induce"; "crouching"; "reprieve";  
              "indigent"; "arrogate"; "coalesce";  
              "temerity"]  
  
let text1 = ["i"; "looked"; "up"; "again"; "at";  
            "the"; "crouching"; "white"; "shape"; "and";  
            "the"; "full"; "temerity"; "of"; "my"; "voyage"]  
  
let test () : bool =  
  equals (findVocab text1 vocab)  
        (set_of_list [ .. ]) ;  
;; run_test "findVocab" test
```

OCaml's = operation may not  
be the right implementation  
for all representations

## 4. Implement the Required Behavior

```
let rec findVocab (text : string list)
                  (vocab : string set) : 'a set =
  begin match text with
  | [] -> empty
  | hd :: tl -> if member hd vocab
                 then add hd (findVocab tl vocab)
                 else findVocab tl vocab
  end
```

- Requires set creation and membership test

```
let empty : 'a set = ...
let add (x: 'a) (s: 'a set) : 'a set = ...
let member (x: 'a) (s: 'a set) : bool = ...
```

# The set interface in OCaml (a *signature*)

```
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
```

Keyword '**val**' names values that must be defined and their types.

# *A module implements an interface*

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

Name of the module

Signature that it implements

```
module ULSet : SET = struct
  ...
  (* implementations of all the operations *)
  ...
end
```

# 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
```

# 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’ or ‘remove’?
  - How big will the sets need to 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?*

# 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.
- Benefits:
  - **Safety:** The other parts of the program can't break any invariants
  - **Modularity:** It is possible to change the implementation without changing the rest of the program

# Set signature

```
module type SET = sig
```

```
  type 'a set
```

Type declaration has no  
“body” – its representation  
is *abstract*!

```
  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
```

# 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 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

# Another Implementation

```
module ULSet : SET =  
  struct  
    type 'a set = 'a list  
    let empty : 'a set = []  
    ...  
  end
```

A different definition for  
the type set



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

Does this code type check?

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

1. yes
2. no

Answer: yes

Does this code type check?

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

1. yes
2. no

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

If a 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 (caveat: performance may be different)

# 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 not named in the interface

## .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 `F00`
- 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
```

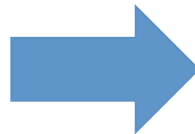
foo.ml

```
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 F00 = sig
  type t
  val z : t
  val f : t -> int
end
```

```
module Foo : F00 = 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
```