Programming Languages and Techniques (CIS120)

Lecture 10

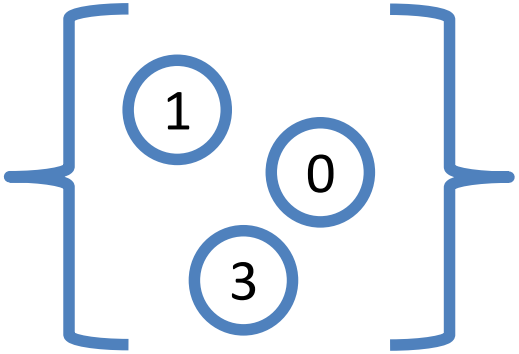
Abstract types: Sets

Chapter 10

Sets as Abstract Types

Mathematical Sets

Mathematical sets are *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

Set operations:

	$S \cup T$	union
	$S \cap T$	intersection

Predicates:	$x \in S$	"x is a member of set S"

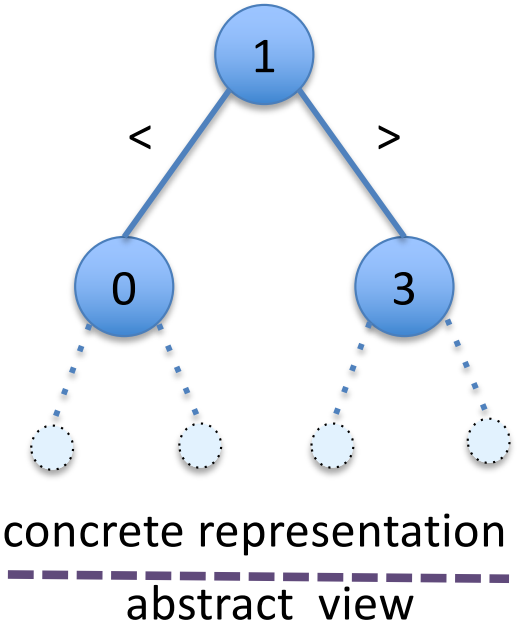
A *set* is a Collection

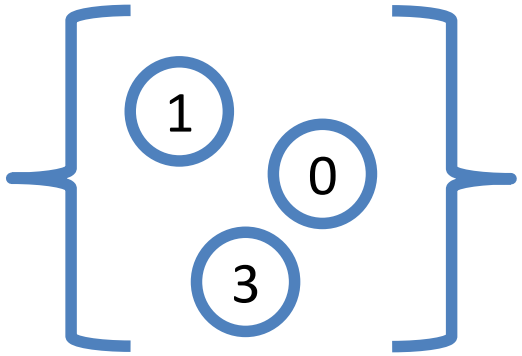
- A set is a *collection* of elements
  - we have operations for creating sets of elements
  - we can ask whether elements are in a set
- Sets show up frequently in code
  - Examples: set of students in a class, set of coordinates in a graph, set of answers to a survey, set of data samples from an experiment, …
- 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…

A Set is an Abstraction

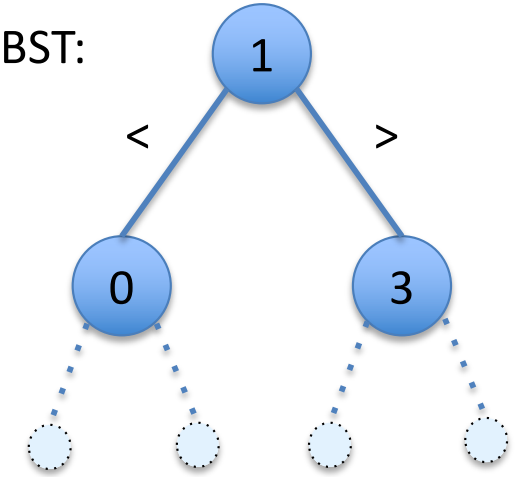
- A BST can *implement (represent)* a *set*
  - *there is a way to represent an empty set (Empty)*
  - *there is a way to list all elements contained in the set (inorder)*
  - *there is a way to test membership (lookup)*
  - *Can define union/intersection (with insert and delete)*
- *BSTs do not have to represent sets*
- *BSTs are not the only way to implement sets*





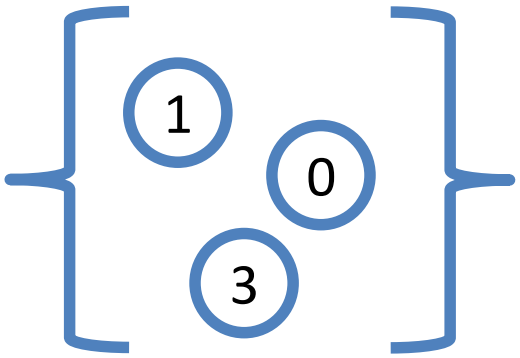
Three Example Representations of Sets

BST:



concrete representation

abstract view

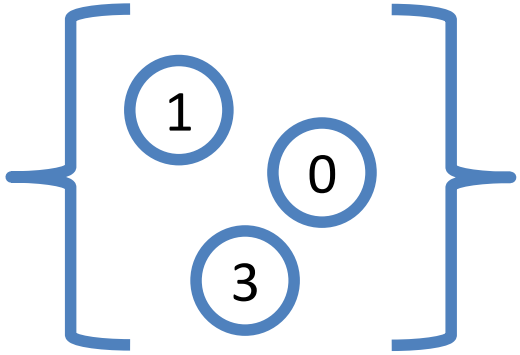


Alternate representation: unsorted linked list.

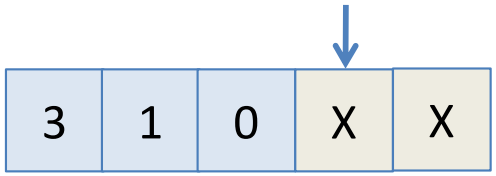
```
3::0::1::[]
```

concrete representation

abstract view

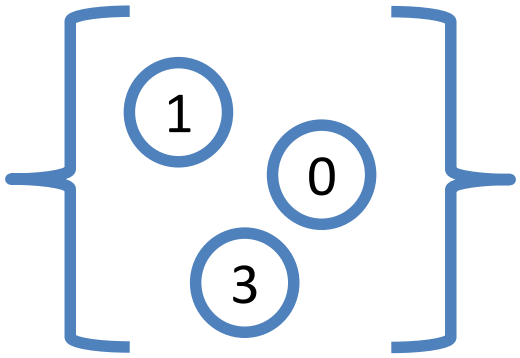


Alternate representation: reverse sorted array with Index of next slot.



concrete representation

abstract view

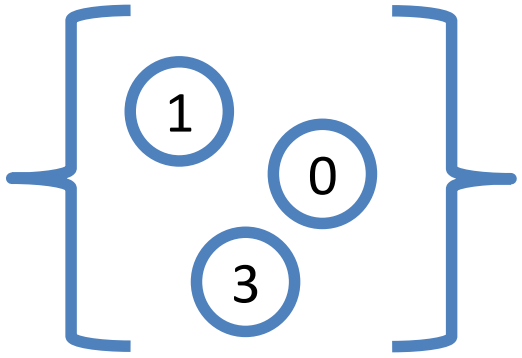


A Set is an Abstract Type

- 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 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 listing of elements
  - Adding elements in a different order doesn't change the listing of elements
- *When we use a set, we can forget about the representation!*

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

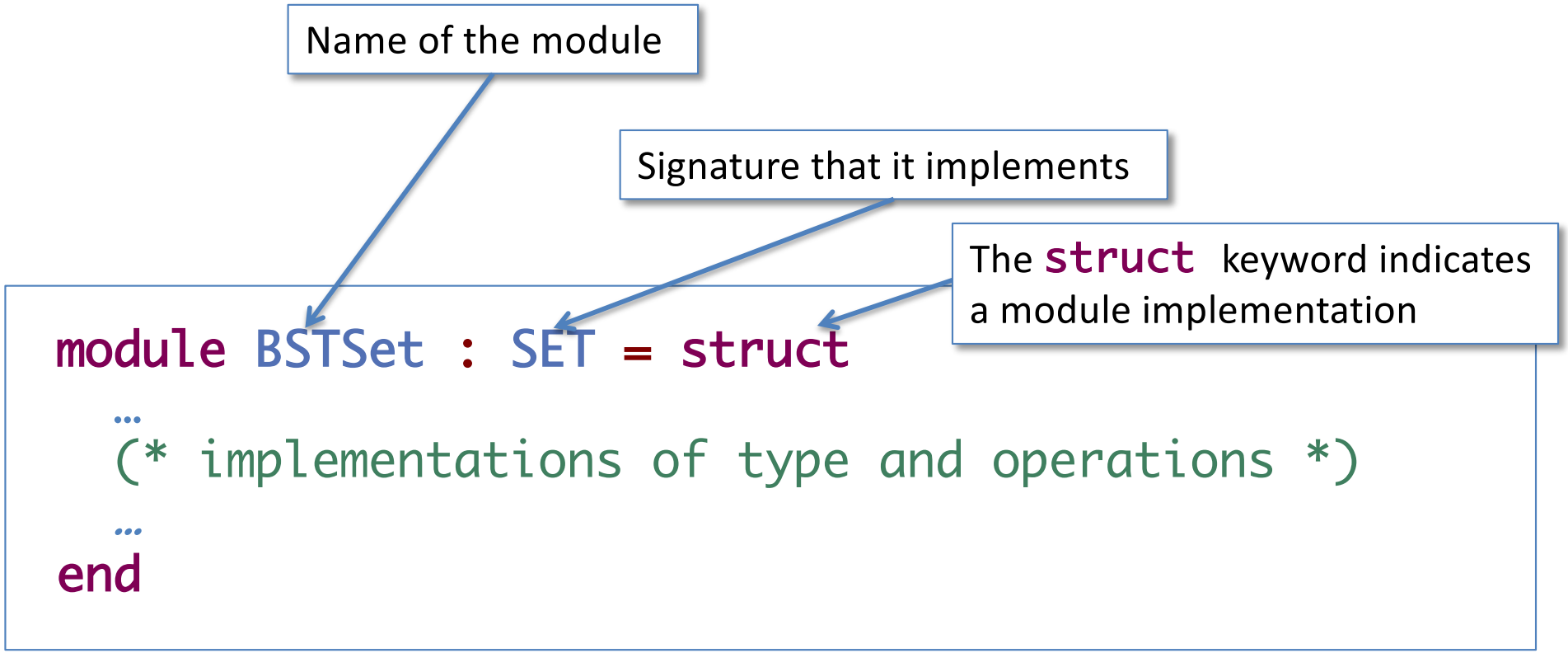
- There are many ways to implement sets
  - lists, trees, arrays, etc.
- *How do we choose which implementation?*
  - Depends on the needs of the application…
  - How often is 'member' used vs. 'add'?
  - How big can the sets be?

- *How do we preserve the invariants of the implementation?*
- 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*

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:



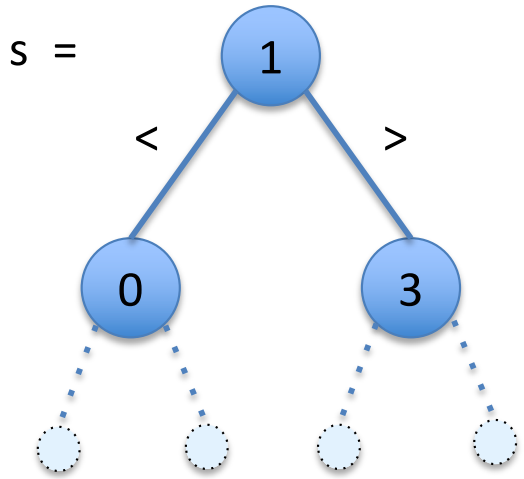
Implement the `set` Module

```
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

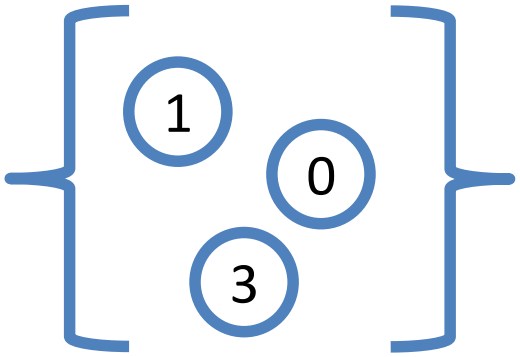
- The implementation must include everything promised by the interface
  - It can contain *more* functions and type definitions (e.g. auxiliary or helper functions) but those cannot be used outside the module
  - The types of the provided implementations must match the signature

Abstract vs. Concrete BSTSet



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 *)
(* Cannot treat a set as a tree  *)
;; 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

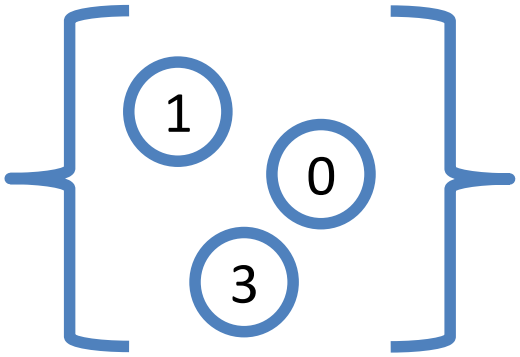
Abstract vs. Concrete ULSet

```
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 *)
(* Cannot treat a set as a list *)
;; open ULSet

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

Client code doesn't change!

Testing (and using) sets

- Use "open" to bring all names defined in the interface into scope
- Any names that were already in scope are shadowed

```
;; 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
```

Testing (and using) sets

- Alternatively, use the "dot" syntax:

  `ULSet.<member>`

- Note: Module names must be capitalized in OCaml
- Useful when two modules define the same operations

```
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
```

```
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

```
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

```
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

```
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

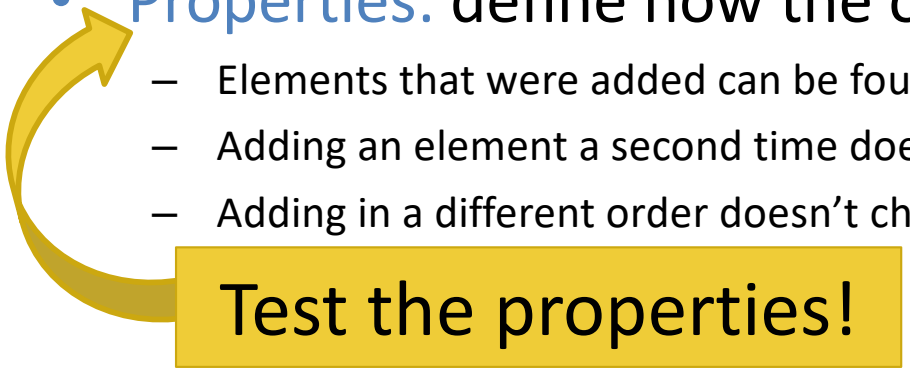
No: the BSTSet operations preserve the BST invariants. there is no way to construct a non-BST tree using the interface.

Completing ULSet

See sets.ml

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



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.equals` is *not* the same as =.

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 is a BST
- Benefits
  - **Safety**: The other parts of the program can't violate invariants, which would cause bugs
  - **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 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