# CS 3110

# Specification

Prof. Clarkson
Fall 2016

Today's music: "A Fifth of Beethoven" by Walter Murphy

# Review

**Previously in 3110:**

- architecture and design of large programs

**Today:**

- specification
  - for clients
  - for implementers

# Question

Think about `java.util` (or some other library you've used frequently). How do you usually come to understand the functionality it provides?

A. **By example:** I search until I find code using the library, then tweak the code to do what I want.

B. **By tutorial:** I read the library's tutorial to understand how it works, then I write code inspired by it.

C. **By documentation:** I read the official documentation for functions, classes, etc., in the library, then I write code from scratch.

D. **By implementation:** I download the source code for the library, read it, then write my own code.

E. I never really understood `java.util`.

# What if you had to read the implementation?

```
let rec sort n l =
    match n, l with
    | 2, x1 :: x2 :: _ ->
      if cmp x1 x2 <= 0 then [x1; x2] else [x2; x1]
    | 3, x1 :: x2 :: x3 :: _ ->
      if cmp x1 x2 <= 0 then begin
        if cmp x2 x3 <= 0 then [x1; x2; x3]
        else if cmp x1 x3 <= 0 then [x1; x3; x2]
        else [x3; x1; x2]
      end else begin
        if cmp x1 x3 <= 0 then [x2; x1; x3]
        else if cmp x2 x3 <= 0 then [x2; x3; x1]
        else [x3; x2; x1]
      end
    | n, l ->
      let n1 = n asr 1 in
      let n2 = n - n1 in
      let l2 = chop n1 l in
      let s1 = rev_sort n1 l in
      let s2 = rev_sort n2 l2 in
      rev_merge_rev s1 s2 []
…
```

# Example specification

```
val sort : ('a -> 'a -> int) -> 'a list -> 'a list
```

Sort a list in increasing order according to a comparison function. The comparison function must return 0 if its arguments compare as equal, a positive integer if the first is greater, and a negative integer if the first is smaller (see Array.sort for a complete specification). For example, `compare` is a suitable comparison function. The resulting list is sorted in increasing order. `List.sort` is guaranteed to run in constant heap space (in addition to the size of the result list) and logarithmic stack space.

**Exercise:** take 2 minutes. Feel free to talk with someone near you. Identify any preconditions and postconditions.

# Example specification

- **One-line summary of behavior:** *Sort a list in increasing order according to a comparison function.*
- **Precondition:** *The comparison function must return 0 if its arguments compare as equal, a positive integer if the first is greater, and a negative integer if the first is smaller (see Array.sort for a complete specification). For example, compare is a suitable comparison function.*
- **Postcondition:** *The resulting list is sorted in increasing order.*
- **Promise about efficiency:** *List.sort is guaranteed to run in constant heap space (in addition to the size of the result list) and logarithmic stack space.*

# Question

What grade would you give the List.sort specification?

A. It provides pre- and postconditions. They are specific enough for me to understand how to use the function as a client. They do not contain irrelevant details or vague descriptions.
B. Parts of the specification are hard to understand. Some details are missing, or some parts are vague.
C. The specification is confusing or just plain wrong.

# Specifications

A **specification** is a contract between an implementer of an abstraction and a client of an abstraction

- Describes behavior of abstraction
- Clarifies responsibilities
- Makes it clear who to blame

An implementation **satisfies** a specification if it provides the described behavior

Many implementations can satisfy the same specification

- Client has to assume it could be any of them
- Implementer gets to pick one

# Specification

Writing good specs is hard:

- the language and compiler do not demand it
- if you're coding only for yourself, neither do you

Reading specs is also hard:

- requires close attention to detail

# ABSTRACTION BY SPECIFICATION

# Abstraction by specification

- Document behavior of function
  - Summary of behavior
  - Pre- and post-conditions
  - Sample usages
- Specification is a kind of abstraction:
  - Forgetting about details
  - Use documentation to reason about behavior instead of having to read implementation

# Benefits of abstraction by specification

- **Locality:** abstraction can be understood without needing to examine implementation
  - critical in implementing large programs
  - also important in implementing smaller programs in teams
- **Modifiability:** abstraction can be reimplemented without changing implementation of other abstractions
  - update standard libraries without requiring world to rewrite code
  - performance enhancements: write the simple slow thing first, then improve bottlenecks as necessary (cf. A3!)

# Good specifications

- **Sufficiently restrictive:** rule out implementations that wouldn't be useful to clients
  - common mistakes: not stating enough in preconditions, failing to identify when exceptions will be thrown, failing to specify behavior at boundary cases
- **Sufficiently general:** do not rule out implementations that would be useful to clients
  - common mistakes: writing operational specifications instead of definitional (saying how, not what), stating too much in a postcondition

Goal is to write specifications that balance being restrictive and general

# When to write specifications

- **During design:**
  - as soon as a design decision is made, document it in a specification
  - posing and answering questions about behavior clarifies what to implement
- **During implementation:**
  - update specification during code revisions
  - a specification becomes obsolete only when the abstraction becomes obsolete

# Audience of specification

- **Clients**
  - Spec informs what they must guarantee (preconditions)
  - Spec informs what they can assume (postconditions)
- **Implementers**
  - Spec informs what they can assume (preconditions)
  - Spec informs what they must guarantee (postconditions)

But the spec isn't **enough** for implementers...

# DOCUMENTING DATA ABSTRACTIONS

# Example: sets

```
module type Set = sig
  type 'a set
  val empty : 'a set
  val mem   : 'a -> 'a set -> bool
  val add   : 'a -> 'a set -> 'a set
  val size  : 'a set -> int
end
```

# Sets without duplicates

```
module ListSetNoDup : Set = struct
  (* the list may never have duplicates *)
  type 'a set = 'a list
  let empty = []
  let mem = List.mem
  let add x l =
    if mem x l then l else x :: l
  let size = List.length
end
```

# Sets with duplicates

```
module ListSetDup : Set = struct
  (* the list may have duplicates *)
  type 'a set = 'a list
  let empty = []
  let mem = List.mem
  let add x l = x :: l
  let rec size = function
  | [] -> 0
  | h::t -> size t +
               (if mem h t then 0 else 1 )
end
```

# Compare set implementations

- Both have the same representation type, `'a list`
- But they interpret values of that type differently
  - `[1;1;2]` is {1,2} in **`ListSetDup`**
  - `[1;1;2]` is not meaningful in **`ListSetNoDup`**
  - In both, `[1;2]` and `[2;1]` are {1,2}
- Interpretation differs because they make different assumptions about what values of that type can be:
  - passed into operations
  - returned from operations
- e.g.,
  - `[1;1;2]` can be passed into and returned from **`ListSetDup`**
  - `[1;1;2]` should not be passed into or returned from **`ListSetNoDup`**

# Question

Consider this implementation of *set union* with representation type **`'a list`**:

```
let union l1 l2 = l1 @ l2
```

Under which assumptions about representation type will that implementation be correct?

A. There are no duplicates in lists

B. There could be duplicates in lists

C. Both A and B

D. Neither A nor B

# Question

Consider this implementation of *set union* with representation type **`'a list`**:

```
let union l1 l2 = l1 @ l2
```

Under which assumptions about representation type will that implementation be correct?

A. There are no duplicates in lists

**B. There could be duplicates in lists**

C. Both A and B

D. Neither A nor B

# Representation type questions

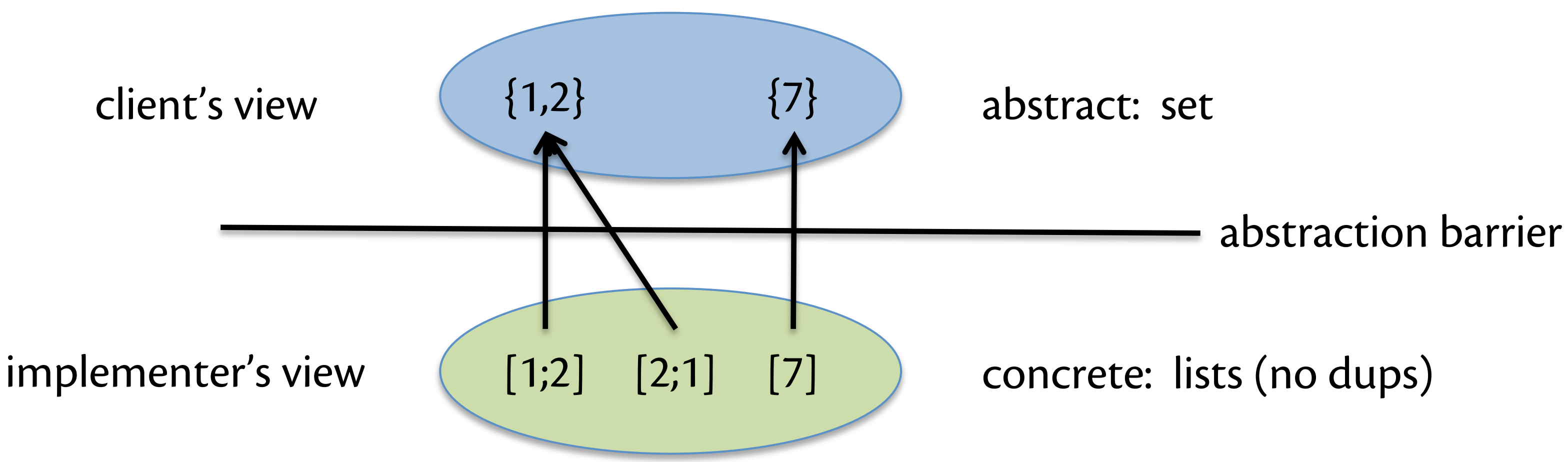
- **Q:** How to *interpret* the representation type as the data abstraction?
- **A:** Abstraction function

- **Q:** How to determine which values of representation type are *meaningful*?
- **A:** Representation invariant

# ABSTRACTION FUNCTION

# Abstraction function

- **Abstraction function** (AF) captures designer's intent in choosing a particular representation of a data abstraction
- Not actually an OCaml function, but a mathematical function
- Maps *concrete values* to *abstract values*



# AF properties

- *Many-to-one*: many values of concrete type can map to same value of abstract type
  - `[1;2]` maps to {1,2}, as does `[2;1]`
- *Partial*: some values of concrete type do not map to any value of abstract type
  - `[1;1;2]` (in no dups) does not map to any set

# Documenting AFs

```
module ListSetNoDup : Set = struct
  (* AF: the list [a1; ...; an] represents
   *   the set {a1,...,an}.  [] represents
   *   the empty set. *)
  type 'a set = 'a list
  ...
end
module ListSetDup : Set = struct
  (* AF: the list [a1; ...; an] represents
   *   the smallest set containing the
   *   elements a1, ..., an.  [] represents
   *   the empty set. *)
  type 'a set = 'a list
  ...
end
```

# Documenting AFs

- You might write:
  - `(* Abstraction Function: `*`comment`*` *)`
  - `(* AF: `*`comment`*` *)`
- You write it FIRST
  - It's the number one decision you have to make while implementing a data abstraction
  - It gives meaning to representation
  - It dictates what fields are necessary in an object, or what values are necessary in a module

# Implementing AFs

- Mostly you don't
  - Would need to have an OCaml type for abstract values
  - If you had that type, you'd already be done...
- But sometimes you do something similar:
  - **`string_of_X`** or **`to_string`** or **`format`**
  - quite useful for debugging

# Upcoming events

- [last night] A3 released
- [next Wed] A3 due

*This is abstract.*

**THIS IS 3110**