
CMSC 330: Organization of Programming Languages

Basic OCaml Modules

Modules

- Defining all functions/variables/etc. at the “top-level” is not good software engineering practice
 - Names in distinct libraries/components could conflict
- Instead: Use **modules** to **group associated types, functions, and data together**
 - Avoid polluting the top-level with unnecessary stuff
- For lots of sample modules, see the OCaml standard library, e.g., **List**, **Str**, etc.

Creating A Module In OCaml

```
module IntSet =
  struct
    type set = Empty | Ins of int * set

    let empty = Empty

    let isEmpty s = (s = Empty)

    let insert s i = Ins(i,s)

    let rec contains s i =
      match s with
      | Empty      -> false
      | Ins(j,r)   -> i = j || (contains r i)
  end;;
```

Creating A Module In OCaml (cont.)

```
module IntSet =
  struct
    type set = ...
    let empty = ...
    let isEmpty = ...
    let insert = ...
    let contains = ...
  end;;

# empty;;
Error: Unbound value empty
# IntSet.empty;;
- : IntSet.set = IntSet.Empty
# IntSet.contains IntSet.empty 1;;
- bool = false
# open IntSet;; (* add IntSet names to curr scope *)
# empty;;
- : IntSet.Empty (* now defined *)
```

Module Signatures

Entry in signature

Supply function types

```
module type FOO =
  sig
    val add : int -> int -> int
  end;;
module Foo : FOO =
  struct
    let add x y = x + y
    let mult x y = x * y
  end;;
Foo.add 3 4;;      (* OK *)
Foo.mult 3 4;;    (* not accessible *)
```

Give type to module

Module Signatures (cont.)

- Convention: **Signature names in all-caps**
 - This isn't a strict requirement, though
- Items can be **omitted from a module signature**
 - This provides the ability to **hide** values
- The **default signature for a module hides nothing**
 - This is what OCaml gives you if you just type in a module with no signature at the top-level

Abstraction = Hiding

- Signatures **hide** module implementation details
 - Why do that? Doesn't that reduce flexibility?
- This is good software engineering practice
 - Ensures data structure **invariants maintained**
 - clients can't construct arbitrary data structures, only ones our module's functions create.
 - E.g., for a BST module, can be sure that clients will not make tree values that violate the BST ordering
 - Facilitates **code collaboration**
 - Write code to the interface as implementation worked out
 - Clients do not rely **details that may change**
 - Changing set representation later won't affect clients

Abstract Data Types

Idea: Hide data value's internal representation from its clients

Invented by **Barbara Liskov** in the **CLU** programming language

- Professor at MIT since 1971

Won **Turing Award** for ADTs and other contributions in 2008

http://amturing.acm.org/award_winners/liskov_1108679.cfm



Abstract Data Types In OCaml Sigs

```
module type INT_SET =
  sig
    type set      (* abstract/hidden *)
    val empty    : set
    val isEmpty  : set -> bool
    val insert   : set -> int -> set
    val contains : set -> int -> bool
  end;;

module IntSet : INT_SET =
  struct
    type set = Empty | Ins of int * set
    ...
    let insert s i = Ins(i,s)
  end
```

- The definition of `set` hidden to `IntSet` clients

Quiz 1: Evaluation on ADTs

```
# IntSet.empty;;  
- : IntSet.set = <abstr> (* OCaml won't show impl *)  
# IntSet.Empty;;  
Unbound Constructor IntSet.Empty  
# IntSet.isEmpty (IntSet.insert IntSet.empty 0);;  
- : bool = false  
# open IntSet;;  
(* doesn't make anything abstract accessible *)
```

```
# IntSet.insert IntSet.empty 0;;
```

A. - : ISet.set = <abstr>

B. Type Error

C. - : ISet.Ins (0, ISet.Empty)

Quiz 1: Evaluation on ADTs

```
# IntSet.empty;;  
- : IntSet.set = <abstr> (* OCaml won't show impl *)  
# IntSet.Empty;;  
Unbound Constructor IntSet.Empty  
# IntSet.isEmpty (IntSet.insert IntSet.empty 0);;  
- : bool = false  
# open IntSet;;  
(* doesn't make anything abstract accessible *)
```

```
# IntSet.insert IntSet.empty 0;;
```

A. - : ISet.set = <abstr>

B. Type Error

C. - : ISet.Ins (0, ISet.Empty)

Multiple representations

```
module IntSetBST : INT_SET =
  struct
    type set = Tip | Bin of int * set * set
    ...
    let rec insert s i =
      match s with
      | Tip -> Bin(i,Tip,Tip)
      | Bin(j,l,r) ->
        if i = j then s
        else if i < j then Bin(j,insert l i,r)
        else Bin(j,l,insert r i)
    end
```

- Now **set** is a binary search tree (why?)

Quiz 2: Mixing ADTs?

```
# IntSetBST.empty;;  
- : IntSetBST.set = <abstr>  
# IntSetBST.insert IntSetBST.empty 0;;  
- : IntSetBST.set = <abstr>  
# IntSetBST.contains IntSetBST.empty 0;;  
- : bool = false
```

```
# IntSet.insert IntSetBST.empty 0;;
```

A. - : IntSet.set = <abstr>

B. - : IntSetBST.set = <abstr>

C. *Type Error*

D. - : IntSetBST.Ins (0, IntSet.Empty)

Quiz 2: Mixing ADTs?

```
# IntSetBST.empty;;  
- : IntSetBST.set = <abstr>  
# IntSetBST.insert IntSetBST.empty 0;;  
- : IntSetBST.set = <abstr>  
# IntSetBST.contains IntSetBST.empty 0;;  
- : bool = false
```

```
# IntSet.insert IntSetBST.empty 0;;
```

A. - : IntSet.set = <abstr>

B. - : IntSetBST.set = <abstr>

C. Type Error

D. - : IntSetBST.Ins (0, IntSet.Empty)

Different ADTs are ... different

- The **set** type of **IntSet** and **IntSetBST** are similar, but not interchangeable
 - Both are om modules that match the **INT_SET** signature
 - But it is not safe to mix them – they have different representations.
- This distinction is enforced by the type system
 - the **set** **type** is an **abstract type**
 - the instances of **modules** having the **INT_SET** signature (which has an abstract type) are called **abstract data types** (ADTs)

Other Module Systems

- How OCaml's approach compare to modularity in...
 - Java?
 - C?
 - Ruby?

Modules In Java

- Java **classes** are like modules
 - Provide implementations for a group of functions
 - But classes can also
 - Instantiate objects
 - Inherit attributes from other classes
- Java **interfaces** are like module signatures
 - Defines a group of functions that may be used
 - Implementation is hidden
 - But: **Objects and modules/ADT not the same**
 - Future lecture topic!

Modules In C

- **.c** files are like modules
 - Provides implementations for a group of functions
- **.h** files are like module signatures
 - Defines a group of functions that may be used
 - Implementation is hidden
- Usage is not enforced by C language
 - Can put C code in .h file



Modules In Ruby

- Ruby explicitly supports modules
 - Modules defined by `module ... end`
 - Modules cannot
 - Instantiate objects
 - Derive subclasses

```
puts Math.sqrt(4)      # 2
puts Math::PI          # 3.1416

include Math           # open Math
puts sqrt(4)           # 2
puts PI                # 3.1416
```