CSE216 Programming Abstractions

Modular Abstractions

YoungMin Kwon

Modular Abstraction

- Modular design of a car

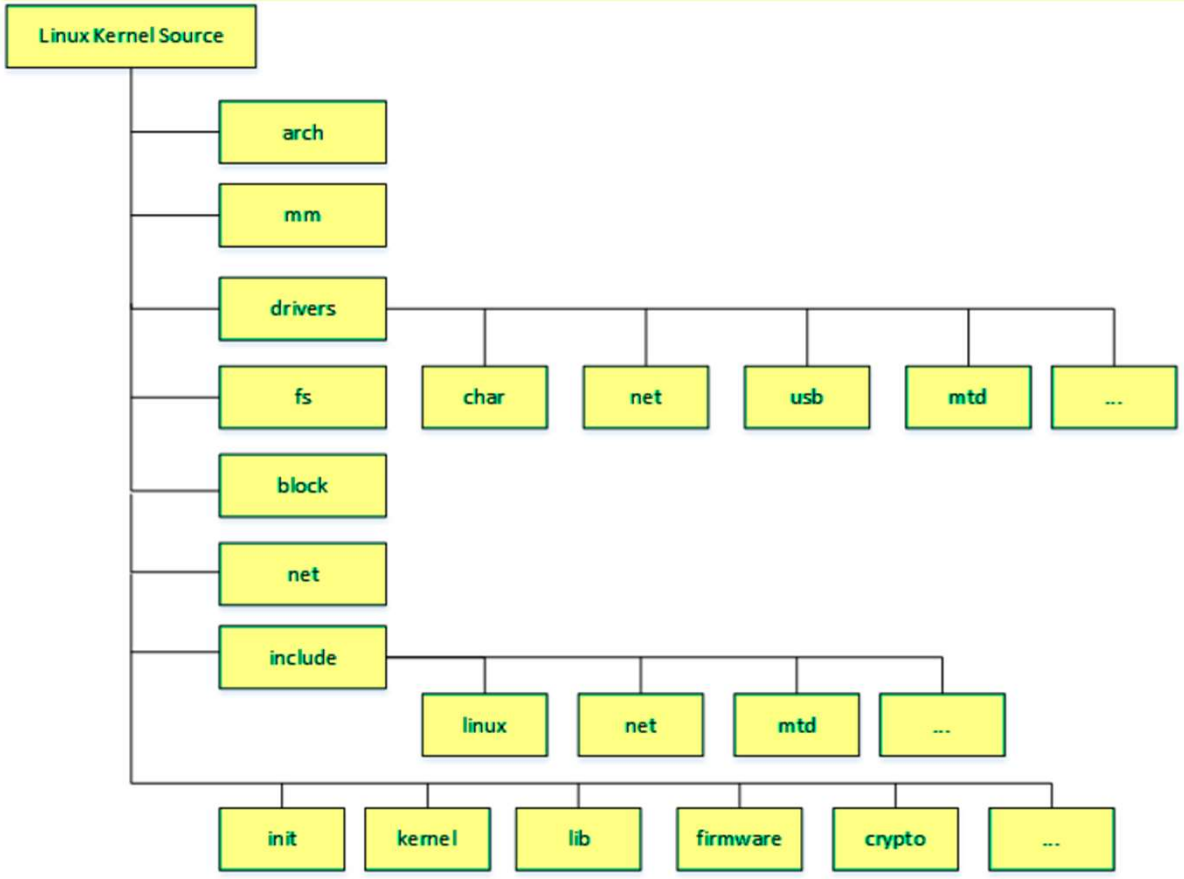
Modules

- To cope with complexities of large programs
  - Procedural abstractions
  - Data abstractions
  - Need organizational principles

- Structure large systems into modules
  - Modules: coherent parts that can be developed and maintained separately

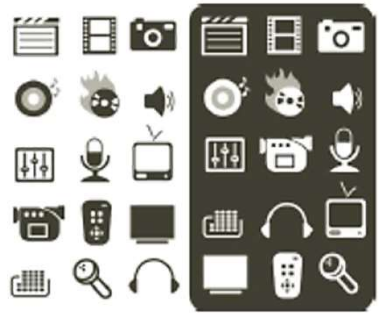
Modules

- Real application programs are huge
  - Often developed by multiple teams
  - Not easy for a single person to grasp every details



Modules

- Modular programming
  - A program is divided into related pieces
  - Each piece can be developed and maintained separately



- Abstraction
  - Modules are abstracted by interfaces
  - The whole program is correct if its modules are correct

Modules

- Modules in OCaml
  - Divide a program into files of conceptual units
  - Each source file $\rightarrow$ a module
  - Module name is derived from the file name
  - A module is similar to a Java class but with static fields and static methods only
    - Not like object instances

Modules

- Modules provide
  - Name space
    - *add* in Complex_number module is different than *add* in Rational_number module
  - Abstraction
    - Reveals the most essential part of the program through interfaces
  - Code reuse
    - Provide the features without having to copy and paste the code

Signatures

- Interface
  - A module defined by *<filename>.ml* can be constrained by a signature file called *<filename>.mli*
    - Without the *.mli* file, all names in the *.ml* file will be exposed
  - Interface, signature, and module type mean the same thing

Signatures

- In mli files
  - Use val keyword to specify values in a signature

    val <identifier> : <type>

  - Use type keyword to specify types

    type <identifier> = <type>

  - To hide the actual type used in ml files, abstract it by specifying only its name (called abstract type)

    type <identifier>

Signatures and Abstract Types

Example: cmplx.mli

```
type complex

(*constructors*)
val rect:  float -> float -> complex
val polar: float -> float -> complex

(*accessors*)
val re:  complex -> float
val im:  complex -> float
val mag: complex -> float
val ang: complex -> float

(*complex arithmetic package*)
val add: complex -> complex -> complex
val sub: complex -> complex -> complex
val mul: complex -> complex -> complex
val div: complex -> complex -> complex

val to_str: complex -> string
```

Exposing complex as an abstract type

Only these names will be exposed in Cmplx module

Signatures and Abstract Types

Example: cmplx.ml

Providing the concrete type

```
type complex = Rect  of float * float
             | Polar of float * float

(*constructors*)
let rect  r i = Rect  (r, i)
let polar m a = Polar (m, a)

(*accessors*)
let re = function
    | Rect  (r, i) -> r
    | Polar (m, a) -> m *. cos(a)
let im = function
    | Rect  (r, i) -> i
    | Polar (m, a) -> m *. sin(a)
let mag = function
    | Rect  (r, i) -> sqrt (r *. r +. i *. i)
    | Polar (m, a) -> m
let ang = function
    | Rect  (r, i) -> atan2 i r
    | Polar (m, a) -> a
```

Signatures and Abstract Types

Example: cmplx.ml

```
(*complex arithmetic package*)
let add a b = rect  (re a +. re b) (im a +. im b)
let sub a b = rect  (re a -. re b) (im a -. im b)
let mul a b = polar (mag a *. mag b) (ang a +. ang b)
let div a b = polar (mag a /. mag b) (ang a -. ang b)

let to_str = function
    | Rect  (r, i) -> Printf.sprintf "%f + %fi" r i
    | Polar (m, a) -> Printf.sprintf "%f / %f" m a
```

To Compile Module Files

- To compile modules

```
$ ocamlc cmplx.mli cmplx.ml

$ ls cmplx.*
…
cmplx.cmi   cmplx.cmo   cmplx.ml    cmplx.mli
```

- To use compiled modules

```
# #load "cmplx.cmo";;

# let a = Cmplx.rect 1. 2.;;
val a : Cmplx.complex = <abstr>

# open Cmplx;;
# let b = rect 2. 3.;;
val b : Cmplx.complex = <abstr>
# to_str (add a b);;
- : string = "3.000000 + 5.000000i"
#
```

Signatures and Abstract Types

Example: cmplx_app.ml

open to omit prefixing Cmplx to the module elements

Module name is Cmplx with capital C

```
open Cmplx

(*a program using complex arithmetic package*)
let to_rect  c = Cmplx.rect (Cmplx.re c) (Cmplx.im c)
let to_polar c = polar (mag c) (ang c)

let a = rect 3. 4.
let b = rect 1. 2.
let c = div a b
let _ = Printf.printf "%s\n" (to_str (to_rect c))
let _ = Printf.printf "%s\n" (to_str (to_polar c))

let _ = exit 0 (*to terminate the process*)
```

To terminate the process

To Build an Executable File

- To compile

The order of the file names is important

```
$ ocamlc -o cmplx.exe cmplx.mli cmplx.ml cmplx_app.ml
```

Drop the suffix .exe in Linux or MAC systems

- Output files

```
$ ls cmplx*.*
cmplx.ml        cmplx.mli       cmplx.cmi       cmplx.cmo
cmplx.exe       cmplx_app.cmi   cmplx_app.cmo   cmplx_app.ml
```

  - *.cmi files: compiled interface files
  - *.cmo files: bytecode objects (like .class in java)

How to Compile

- To compile individual files

```
> ocamlc -c cmplx.ml
File "cmplx.ml", line 1:
Error: Could not find the .cmi file for interface cmplx.mli.
```

  - Compile mli files first

```
> ocamlc -c cmplx.mli
> ocamlc -c cmplx.ml
> ocamlc -c cmplx_app.ml
```

- To link

```
> ocamlc -o cmplx.exe cmplx.cmo cmplx_app.cmo
```

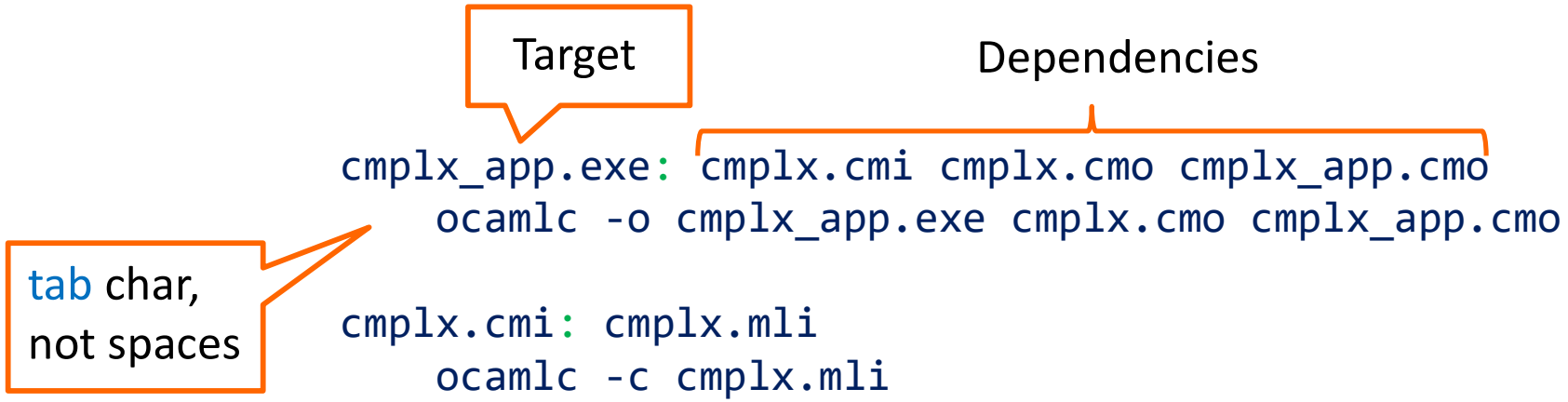
How to Compile

- To make an archive file
  - A .cma file is an archive of many related .cmo files
    - Like a .jar file is an archive of many .class files in java

```
> ocamlc -a -o cmplx.cma cmplx.cmo

> ocamlc -o cmplx.exe cmplx.cma cmplx_app.cmo

> cmplx.exe
2.200000 + -0.400000i
2.236068 / -0.179853

> ocaml cmplx.cma cmplx_app.cmo
2.200000 + -0.400000i
2.236068 / -0.179853
```

Compile with make

- Build with make tool
  - Write Makefile as below and run make
    - make <target> like make cmplx.cmo will build the target




```
cmplx_app.exe: cmplx.cmi cmplx.cmo cmplx_app.cmo
    ocamlc -o cmplx_app.exe cmplx.cmo cmplx_app.cmo

cmplx.cmi: cmplx.mli
    ocamlc -c cmplx.mli

cmplx.cmo: cmplx.ml
    ocamlc -c cmplx.ml

cmplx_app.cmo: cmplx_app.ml
    ocamlc -c cmplx_app.ml
```

Compile with make (suffix rules)

```
TGT  = cmplx_app.exe
CMIS = cmplx.cmi
CMOS = cmplx.cmo cmplx_app.cmo

RM   = rm   # rm   in Linux, del  in Windows
TRUE = true # true in Linux, cd . in Windows

.SUFFIXES:                      # reset all suffixes
.SUFFIXES: .cmi .cmo .mli .ml # suffixes to consider

.mli.cmi:; ocamlc -c $< -o $@  # how to convert .mli to .cmi
.ml.cmo:;  ocamlc -c $< -o $@  # $< : input file name,
                               # $@ : target file name

$(TGT): $(CMIS) $(CMOS)
    ocamlc -o $@ $(CMOS)

clean:
    $(RM) *.cmi | $(TRUE)
    $(RM) *.cmo | $(TRUE)
```

macro definitions

suffix rules

using macros

true even if $(RM) failed

Loading Compiled Modules

- In the OCaml toplevel
  - #load .cmo or .cma file to load the module

```
# #load "cmplx.cma";;

# Cmplx.to_str (Cmplx.rect 1. 2.);;
- : string = "1.000000 + 2.000000i"
```

  - Use open to skip prefixing the module name

```
# open Cmplx;;

# to_str (rect 1. 2.);;
- : string = "1.000000 + 2.000000i"
```

Module Exercise

```
(*amodule.mli*)
val hello : unit -> unit
val print_list : string list -> unit
```

```
(*amodule.ml*)
open Printf
let message = "Hello"
let hello () = printf "%s\n" message
let print_list lst = List.iter (fun s -> printf "%s\n" s) lst
```

```
(*bmodule.ml*)
open Amodule
let _ = hello ()
let _ = print_list ["what"; "a"; "beautiful"; "day"]
```

Module Exercise

- Compile modules

```
$ ocamlc amodule.mli

$ ocamlc amodule.ml bmodule.ml

$ ls ?module.*

amodule.cmi   amodule.cmo   amodule.ml    amodule.mli
bmodule.cmi   bmodule.cmo   bmodule.ml

$ ocamlc -o hello.exe amodule.cmo bmodule.cmo

$ .\hello.exe
Hello
what
a
beautiful
day
```

Compile with make

```
TGT  = hello.out
CMIS = amodule.cmi
CMOS = amodule.cmo bmodule.cmo

RM   = rm   # rm   in Linux, del  in Windows
TRUE = true # true in Linux, cd . in Windows

.SUFFIXES:                      # reset all suffixes
.SUFFIXES: .cmi .cmo .mli .ml # suffixes to consider

.mli.cmi:; ocamlc -c $< -o $@  # how to convert .mli to .cmi
.ml.cmo:;  ocamlc -c $< -o $@  # $< : input file name,
                               # $@ : target file name

$(TGT): $(CMIS) $(CMOS)
        ocamlc -o $@ $(CMOS)

clean:
        $(RM) *.cmi | $(TRUE)
        $(RM) *.cmo | $(TRUE)
```

Nested Modules

- Modules can have nested modules inside
  - Signature definition

```
module type <name> = sig <signature> end
```

  - Module definition

```
module <name> : <sig name> = struct <implementation> end
```

Nested Module

- Example: stack
  - Signature

```
(*in mystack.ml*)
module type Istack = sig
    type 'a stack

    val empty: unit -> 'a stack
    val is_empty: 'a stack -> bool
    val push: 'a -> 'a stack -> 'a stack
    val peek: 'a stack -> 'a
    val pop:  'a stack -> 'a stack
end
```

'a stack is an abstract type

Nested Module

- Stack: implementation

```
(*in mystack.ml*)
module Stack: Istack = struct
    type 'a stack = Nil | Cons of 'a * 'a stack

    let empty () = Nil
    let is_empty s = (s = Nil)
    let push a s = Cons (a, s)
    let peek s =
        match s with
        | Nil -> assert false
        | Cons (a, _) -> a
    let pop s =
        match s with
        | Nil -> assert false
        | Cons (_, b) -> b
end
```

Nested Module

- Stack: application

```
(*in mystackapp.ml*)
let reverse lst =
    let open Mystack in
    let rec add lst stack =
        match lst with
        | [] -> stack
        | h::t -> add t (Stack.push h stack) in
    let rec del stack =
        if Stack.is_empty stack then
            []
        else
            Stack.peek stack :: del (Stack.pop stack) in
    Stack.empty ()
        |> add lst
        |> del

let _ = List.iter (fun x -> Printf.printf "%d " x)
                  (reverse [1; 2; 3; 4; 5])
```

Nested Module (mutually recursive)

Example: cmplx2.ml

t is an abstract type

```
module type Cmplx = sig
    type t

    (*constructor*)
    val make:  float -> float -> t

    (*accessors*)
    val re:  t -> float
    val im:  t -> float
    val mag: t -> float
    val ang: t -> float

    (*complex arithmetic package*)
    val add: t -> t -> t
    val sub: t -> t -> t
    val mul: t -> t -> t
    val div: t -> t -> t

    (*to string*)
    val to_str: t -> string
end
```

use rec to define mutually recursive modules

Rect is using module Polar

```
module rec Rect : Cmplx = struct
    type t = float * float

    (*constructor*)
    let make r i = (r, i)

    (*accessors*)
    let re  (r, i) = r
    let im  (r, i) = i
    let mag (r, i) = sqrt (r *. r +. i *. i)
    let ang (r, i) = atan2 i r

    (*complex arithmetic package*)
    let add (r1, i1) (r2, i2) = (r1 +. r2, i1 +. i2)
    let sub (r1, i1) (r2, i2) = (r1 -. r2, i1 -. i2)
    let mul c1 c2 =
        let a = Polar.make (mag c1) (ang c1) in
        let b = Polar.make (mag c2) (ang c2) in
        let c = Polar.mul a b in
        (Polar.re c, Polar.im c)
    let div c1 c2 =
        let a = Polar.make (mag c1) (ang c1) in
        let b = Polar.make (mag c2) (ang c2) in
        let c = Polar.div a b in
        (Polar.re c, Polar.im c)

    (*to string*)
    let to_str (r, i) =
        Printf.sprintf "%f + %fi" r i
end
```

use and to define mutually recursive modules

signature name with its abstract type t assigned to float*float

Polar is using module Rect

```
and Polar : (Cmplx with type t = float*float) = struct
    type t = float * float

    (*constructor*)
    let make m a = (m, a)

    (*accessors*)
    let re  (m, a) = m *. cos a
    let im  (m, a) = m *. sin a
    let mag (m, a) = m
    let ang (m, a) = a

    (*complex arithmetic package*)
    let add c1 c2 =
        let a = Rect.make (re c1) (im c1) in
        let b = Rect.make (re c2) (im c2) in
        let c = Rect.add a b in
        (Rect.mag c, Rect.ang c)
    let sub c1 c2 =
        let a = Rect.make (re c1) (im c1) in
        let b = Rect.make (re c2) (im c2) in
        let c = Rect.sub a b in
        (Rect.mag c, Rect.ang c)
    let mul (m1, a1) (m2, a2) = (m1 *. m2, a1 +. a2)
    let div (m1, a1) (m2, a2) = (m1 /. m2, a1 -. a2)

    (*to string*)
    let to_str (m, a) =
        Printf.sprintf "%f / %f" m a
end
```

Nested Module

- To compile cmplx2.ml

```
> ocamlc -c cmplx2.ml
```

- To load cmplx2 module

```
# #load "cmplx2.cmo";;

# Cmplx2.Rect.make 1. 2.;;
- : Cmplx2.Rect.t = (1., 2.)

# open Cmplx2;;

# let a = Rect.make 1. 2.;;
val a : Cmplx2.Rect.t = (1., 2.)

# let b = Rect.make 1. 1.;;
val b : Cmplx2.Rect.t = (1., 1.)

# Rect.to_str (Rect.mul a b);;
- : string = "-1.000000 + 3.000000i"
```

Opening Modules

- Opening a module
  - Adds the contents of the module to the environment where the compiler finds the definition of identifiers

```
# module M = struct let three = 3 end;;
module M : sig val three : int end

# three;;
Characters 0-3:
  three;;
  ^^^^^
Error: Unbound value three

# open M;;
# three;;
- : int = 3
```

Opening Modules

- Local open
  - It affects only to the local scope

```
let print lst =
    let open List in
    let open Printf in
    iter (fun x -> printf "%d\n" x) lst

let print2 lst =
    List.(Printf.(  (*alternative syntax*)
        iter (fun x -> printf "%d\n" x) lst))

let _ = print  [1; 2; 3]
let _ = print2 [1; 2; 3]
```

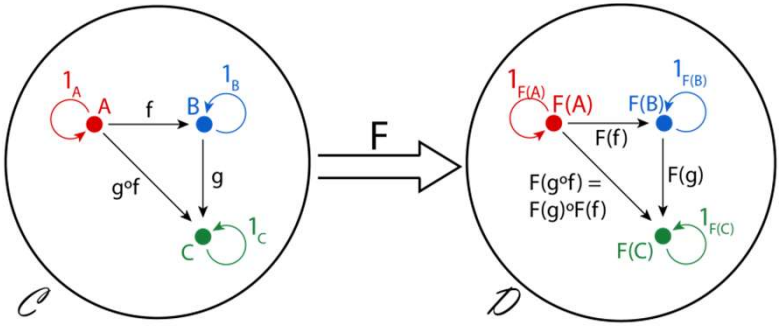
Including Modules

- To extend a module, use *include* directive

```
module Interval = struct
    type t = Interval of int * int
           | Empty

    let create low high =
        if high < low then Empty
                      else Interval (low, high)
end

module Extended_interval = struct
    include Interval

    let contains i x =
        match i with
        | Empty -> false
        | Interval (low, high) -> low <= x && x <= high
end

let i = Extended_interval.create 3 10
let _ = Extended_interval.contains i 4
```

Functor

- Function from a structure to a structure
  - E.g. category
    - A labeled directed graph
    - Associative composition of arrows; existence of identity arrows
    - In category theory, a functor is a mapping between categories



  - In OCaml, a functor is a function from modules to a module

Functor

- Functor definition

```
module F (M: S) = struct … end
```

  - F is the functor name,
  - M is the module parameter, S is the type of M

  - Alternatively

```
module F = functor (M: S) -> struct … end
```

- To construct a module using a Functor

```
module X = F (MImpl)
```

or

```
module X = F (struct … end)
```

Functor

- For multiple parameters

```
module F (M1: S1) … (Mn: Sn) = struct … end

module F = functor (M1: S1) … (Mn: Sn) -> struct … end

module F = functor (M1: S1) -> … ->
           functor (Mn: Sn) -> struct … end
```

- To specify output type

```
module F (M: Si): So = struct <functor definition> end
```

  - Si is the type of M
  - So is the module type of the output

- To construct a module using a Functor

```
module X = F (M1Impl) … (MnImpl)
```

Functor

Trivial Example

Interface

Input sig.

Output sig.

Functor

module One

module Two

module Three

```
module type X_int = sig
    val x: int
end

module Increment (M: X_int): X_int = struct
    let x = M.x + 1
end

module One : X_int = struct
    let x = 1
end

module Two   = Increment (One)
module Three = Increment (Two)

let _ = Printf.printf "%d\n" One.x
let _ = Printf.printf "%d\n" Two.x
let _ = Printf.printf "%d\n" Three.x
```

Functor

- Exercise

```
(*signature and module*)
module type IAdd = sig
  val add: int -> int -> int
end

module Add: IAdd = struct
  let add x y = x + y
end

let _ = Add.add 1 2
```

```
(*functor*)
module type IInc= sig
  val inc: int -> int
end

module FInc (A: IAdd): IInc = struct
  let inc x = A.add 1 x
end

module Inc = FInc (Add)

let _ = Inc.inc 2
```

Example: complex module tester for Rect

```
let equ a b = max (a -. b) (b -. a) < 1e-10
let test_add a b =
    let c = Rect.add a b in
    assert (equ ((Rect.re a) +. (Rect.re b)) (Rect.re c) &&
            equ ((Rect.im a) +. (Rect.im b)) (Rect.im c))
let test_sub a b =
    let c = Rect.sub a b in
    assert (equ ((Rect.re a) -. (Rect.re b)) (Rect.re c) &&
            equ ((Rect.im a) -. (Rect.im b)) (Rect.im c))
let test_mul a b =
    let c = Rect.mul a b in
    assert (equ ((Rect.mag a) *. (Rect.mag b)) (Rect.mag c) &&
            equ ((Rect.ang a) +. (Rect.ang b)) (Rect.ang c))
let test_div a b =
    let c = Rect.div a b in
    assert (equ ((Rect.mag a) /. (Rect.mag b)) (Rect.mag c) &&
            equ ((Rect.ang a) -. (Rect.ang b)) (Rect.ang c))
let test () =
    let a = Rect.make 0.1 0.2 in
    let b = Rect.make 0.2 0.3 in
    test_add a b;    test_sub a b;
    test_mul a b;    test_div a b
```

Functor

Complex module tester

- The test code works only for Rect module
- To make a tester for Polar
  - The code needs to be copied and modified

- Solution
  - Use functor to generate Tester modules

```
module CmplxTester (C: Cmplx) = struct
    let equ a b = max (a -. b) (b -. a) < 1e-10

    let test_add a b =
        let c = C.add a b in
        assert (equ ((C.re a) +. (C.re b)) (C.re c) &&
                equ ((C.im a) +. (C.im b)) (C.im c))
    let test_sub a b =
        let c = C.sub a b in
        assert (equ ((C.re a) -. (C.re b)) (C.re c) &&
                equ ((C.im a) -. (C.im b)) (C.im c))
    let test_mul a b =
        let c = C.mul a b in
        assert (equ ((C.mag a) *. (C.mag b)) (C.mag c) &&
                equ ((C.ang a) +. (C.ang b)) (C.ang c))
    let test_div a b =
        let c = C.div a b in
        assert (equ ((C.mag a) /. (C.mag b)) (C.mag c) &&
                equ ((C.ang a) -. (C.ang b)) (C.ang c))
    let test () =
        let a = C.make 0.1 0.2 in
        let b = C.make 0.2 0.3 in
        test_add a b;          test_sub a b;
        test_mul a b;          test_div a b
end
```

Functor

Complex module tester

- Generate tester modules for Rect and Polar using CmplxTester functor

```
module RectTester  = CmplxTester (Rect)
module PolarTester = CmplxTester (Polar)

let _ = RectTester.test ()
let _ = PolarTester.test ()
```

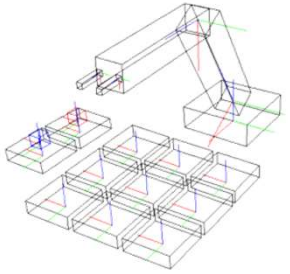
Standard Library Map (dictionary)

- Create a map using a functor

```
module IntMap = Map.Make (struct
    type t = int
    let compare = fun x y -> x - y
end)


let map = IntMap.empty
            |> IntMap.add 1 "one"
            |> IntMap.add 2 "two"

let _ = IntMap.find 1 map
let _ = IntMap.find 2 map
let _ = IntMap.mem 3 map
(*let _ = IntMap.find 3 map*)
```

```
module StrMap = Map.Make (struct
    type t = string
    let compare = fun x y ->
                    String.compare x y
end)

let map = StrMap.empty
            |> StrMap.add "one" 1
            |> StrMap.add "two" 2

let _ = StrMap.find "one" map
let _ = StrMap.find "two" map
let _ = StrMap.mem "three" map
(*let _ = StrMap.find "three" map*)
```

Assignment 5



- Refactor robot using Modules and Functors
  - Download robot_module.zip
  - Update all TODOs and upload the source files in a single zip file
  - make will generate unit test cases and applications
    - vector.out, basis.out, board.out, drawer.out, pose.out, command.out, player.out, game.out
    - app.out, app_kk.out, app_cc.out
    - All unit tests should be passed and the applications should work as expected.
  - Due date 4/23/2024

```
CMAS = unix.cma threads.cma graphics.cma
RM   = rm   # use rm for Linux,   del  for Windows
…

.SUFFIXES:
.SUFFIXES: .cmo .cmi .ml .mli

.mli.cmi:;  ocamlc -c -o $@ $<
.ml.cmo:;   ocamlc -c -thread -o $@ $<

all: vector basis board drawer pose command player game app app_cc app_kk
    echo done.

vector: globals.cmo ivector.cmi vector.cmo test_vector.cmo
    $(eval CMOS = globals.cmo vector.cmo)
    ocamlc -thread $(CMOS) test_vector.cmo -o $@$(OEXT)

basis: vector ibasis.cmi basis.cmo test_basis.cmo
    $(eval CMOS = $(CMOS) basis.cmo)
    ocamlc -thread $(CMOS) test_basis.cmo -o $@$(OEXT)
…
app_kk: game app_kk.cmo
    ocamlc -I threads -thread $(CMAS) $(CMOS) app_kk.cmo -o $@$(OEXT)
```

update CMOS

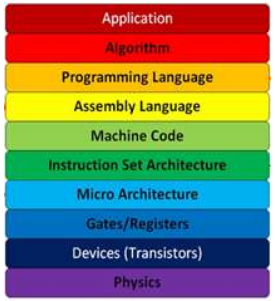
```
(*ivector.mli*)
open Globals
(*TODO: Add a module signature IVect.
  It should expose functions: add, sub, smul, prod, len, rotx, roty, rotz
  hint: use vector type of globals.ml*)

(*vector.ml*)
open Globals
(*TODO: Implement VectImpl module.
        VectImpl implements the signature Ivector.IVect*)

module VectImpl: Ivector.IVect = struct

    (*vector addition*)
    let add (x, y, z) (u, v, w) =
        (*TODO*)

    (*vector subtraction*)
    let sub (x, y, z) (u, v, w) =
        (*TODO*)
```

```
(*test_vector.ml*)

(*-----------------------------------
  Unit Test for Vector 3D
-----------------------------------*)
open Globals

module TestVect = struct
    module Vect = Vector.VectImpl

    let test () =
        Printf.printf("-------------------------------------\n");
        Printf.printf("test vector...\n");
        assert ((3.,5.,7.) = Vect.add (2.,3.,4.) (1.,2.,3.));
        assert ((1.,1.,1.) = Vect.sub (2.,3.,4.) (1.,2.,3.));
        assert ((2.,4.,6.) = Vect.smul 2. (1.,2.,3.));
        assert (20. = Vect.prod (2.,3.,4.) (1.,2.,3.));
        assert (equ 5. (Vect.len (0.,3.,4.)));
        Printf.printf("test vector done\n")
end

(*unit test*)
let _ = TestVect.test ()
```

```
(*basis.ml*)
(*TODO: Implement BasisImpl module.
    BasisImpl takes a module Vect of Ivector.IVect type and
    implements the signature Ibasis.IBasis*)

module BasisImpl (Vect: Ivector.IVect): Ibasis.IBasis = struct
    (*basis scale: scale basis by s (float)*)
    let scale s (o, x, y, z) =
        (*TODO*)

    (*basis translation: translate basis by t (vector)*)
    let translate t (o, x, y, z) =
        (*TODO*)
…

(*test_basis.ml*)
…
module TestBasis = struct

    module Vect = Vector.VectImpl
    module Basis = (*TODO: Basis module using BasisImpl and Vect*)

    let test () =
…
(*unit test*)
let _ = TestBasis.test ()
```

```
(*board.ml*)
(*TODO: Implement BoardImpl functor.
    BoardImpl returns a module of signature Iboard.IBoard*)

…
    (*TODO: refactor: move winner from game to here*)
    (*winner of the board if any; otherwise mark_n*)
    let winner board =
…

(*test_board.ml*)
…
module TestBoard = struct
    module Board = Board.BoardImpl

    (*unit test*)
    let test () =
…
let _ = TestBoard.test ()
```

```
(*idrawer.mli*)
module type IDrawer = sig
    (*open a graphics window*)
    val open_graph: unit -> unit
    (*close the graphics window*)
    val close_graph: unit -> unit
    (*delay*)
    val delay: float -> unit …
end

(*drawer.ml*)
(*TODO: Implement DrawerImpl module.
    DrawerImpl takes modules
        Vect of IVect type, Basis of IBasis type,… in this order
    and implements the signature IDrawer*)
…
    (*open a graphics window*)
    let open_graph () =
        Graphics.open_graph " 800x800"; …
    (*close the graphics window*)
    let close_graph () =
        Graphics.close_graph ()
    (*delay*)
    let delay sec =
        Thread.delay sec …
```



Can be bad for testing

```
(*pose.ml*)
(*TODO: Implement PoseImpl functgor.
    PoseImpl takes modules
        Vect of IVect type, Basis of IBasis type,… in this order
    and returns a module of signature IPose*)


(*test_pose.ml*)
…
module TestPose = struct
    module Vect  = Vector.VectImpl
    module Basis = (*TODO: build Basis using BasisImpl and Vect*)
    module Board = Board.BoardImpl
    module Pose  = (*TODO: build Pose using PoseImpl, Vect, Basis, Board*)
…
```

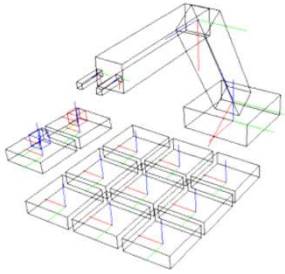
```
(*command.ml*)
(*TODO: Implement CommandImpl functor.
    CommandImpl takes modules
        Pose of IPose type, Drawer of IDrawer type,… in this order
    and returns a module of signature ICommand*)
…
    let rot_joint pose joint ang step =
    (*TODO: implement this method
        - on each step, draw the robot and the board
        - refactor: wait for 50ms by calling
              Drawer.delay 0.05 instead of
              Thread.delay 0.05
        - then either return pose or rotate more
          *)
…
```

```
(*test_command.ml*)
…
module TestCommand = struct
    module Vect  = Vector.VectImpl

    (*TODO: build Basis using BasisImpl and Vect*)
    module Basis =

    module Board = Board.BoardImpl

    (*TODO: build Pose using PoseImpl, Basis, Board*)
    module Pose  =

    (*TODO: build MockDrawer of IDrawer type that simply returns the unit
            from all of its functions without doing anything*)
    module MockDrawer

    (*TODO: build module Command using CommandImpl, Pose, MockDrawer, Board*)
    module Command =
…
```

```
(*player_com.ml*)
(*TODO: Implement PlayerCom functor.
    PlayerCom takes a module Board of Iboard.IBoard type
    and returns a module of signature Iplayer.IPlayer*)

(*player_key.ml*)
(*TODO: Implement PlayerKey functor.
    PlayerKey takes a module Board of Iboard.IBoard type
    and returns a module of signature Iplayer.IPlayer*)

(*test_player_com.ml*)
module TestPlayerCom = struct
    module Board  = Board.BoardImpl
    module Player = (*TODO: build a module for a computer player*)
…       (*play a game with computer*)
        board |> fun b -> Board.chg_mark b 0 mark_o
              |> fun b -> play b
              |> fun i -> assert(1 = i); Board.chg_mark b i mark_x
              |> fun b -> Board.chg_mark b 4 mark_o
              |> fun b -> play b
              |> fun i -> assert(8 = i); Board.chg_mark b i mark_x
              |> fun b -> Board.chg_mark b 3 mark_o
              |> fun b -> play b
              |> fun i -> assert(5 = i); Board.chg_mark b i mark_x
              |> fun b -> Board.chg_mark b 6 mark_o
              |> fun b -> assert(mark_o = Board.winner b);
```

```
(*game.ml*)
(*TODO: Implement GameImpl functor.
    GameImpl takes modules
        Board of IBoard type, Drawer of IDrawer type, …
        PlayerO of IPlayer type PlayerX of IPlayer type in this order
    and returns a module of signature IGame*)

(*test_player_game.ml*)
module TestGame = struct
    module Board = Board.BoardImpl
    (*TODO: build MockDrawer of IDrawer type that simply returns the unit
            from all of its functions without doing anything*)
    module MockDrawer
    module MockCommand: Icommand.ICommand = struct
        let mark basis (pose, board) mrk i =
            Board.chg_mark board i mrk |> fun b ->
                Board.print_board b;
                (pose, b)
    end
    (*TODO: build a module for a computer player*)
    module Player =
    (*TODO: build Game using GameImpl, Board, MockDrawer, MockCommand,
        and two Players: it is a game between two computer players*)
    module Game   =
```

Application



- Applications
  - app.out: game between Keyboard and Computer
  - app_cc.out: game between two Computers
  - app_kk.out: game between two Keybaords

- Functor
  - By choosing different Player modules, we can easily make different applications
  - By providing mock modules, we can test modules without having the actual modules