Principles Of Programming Languages

--

Functional Programming
Ideas

- Imperative programming: modify variables to write algorithms
- Functional programming: evaluate expressions / functions
  - Close to 'math' notation
    - $f(x) = \cos x \sin x$
      - Note: juxtaposition (no '()', context decides)
  - All is function, function is all
    - No global variables
    - No 'heap' to allocate things in
  - High level
    - No low level implementation details
Principles

• Compute result based on function arguments only
  – No side effects allowed and not allowed to read global state, nor state of other functions
    • Makes programs easier to understand (for compiler+programmer)
    • Most FP languages do not even have local variables..
  – When/how/much a function is called: not important
    • “Referential Transparency”
      • Can replace function call by function result always

Example:

\[ f(x) + f(x) \equiv 2 \times f(x) \]

is always true for functional languages, not so for imperative languages
Lambda Calculus

- Define functions using parameters, body
  - Macro-like function evaluation
    - Alpha rename = rename unbound variables
  - Functions may not have side-effects
    - $\lambda x. x + 2$
    - Means that functions can be evaluated multiple times
      - No calling convention needed: simple variable substitution enough
  - Functions can have arguments
    - $(\lambda x. x + 2) 4 \Rightarrow 4 + 2 \Rightarrow 6$
  - Functions can be passed to other functions
    - $(\lambda x. x + 2) (\lambda y. y * y) 4$
  - Functions may have multiple parameters
    - $(\lambda x y. x * y + 2) (4+3) == (\lambda y. (4+3) * y + 2)$
Work-arounds

• Do you really need global variables?
  – No: can use arguments to maintain state
  – No: *read-only* globals *still* allowed

```c
int tmp;

int foo(int a) {
    tmp = tmp + a * 3;
    return tmp + 5;
}

(int, int) foo(int a, int state) {
    return (state + a * 3 + 5), (state + a * 3);
}
```
Work-arounds

• But what about data-structures (arrays, lists, etc) ?
  - In general: create copies of each call-argument
  - Of course: performance losses
    • Use diff/patch
      - Each modification creates a 'diff' over a read-only input argument
    • Copy-on-write
      - Create a copy only upon modification

```java
void sum(int[] sub, int index, int[] array) {
    for i = index * N .. (index+1) * N
        sub[i] += array[i]
}
```

```java
int[] sum(int index, int[] array) {
    int[] s = new int[N];
    int k = 0;
    for i = index * N .. (index+1) * N
        s[k++] = array[i];
    return s;
}
```
Work-arounds

- How to do Input/Output?
  - They are the ultimate side-effect!
    - Temporarily break referential transparency
    - Model I/O using monads (later)
    - Functions that do I/O (recursively) have an I/O return type that is managed specially
    - Use lazy-evaluation (later)

- Pure vs Inpure
  - 'pure' functional --> all functions are referential transparent
  - Impure --> some still imperative / imperative can infect functional functions
Example: Miranda

- Functions have both a prototype and a body

\[
fahrenheit\_to\_celcius :: \text{num} \rightarrow \text{num}\\
fahrenheit\_to\_celcius\ f = ((f - 32) \times 5) / 9
\]

|| Type: 1 number results in 1 number
|| Code
### Examples

- recursion:

<table>
<thead>
<tr>
<th>// Math notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>fac(x) = {</td>
</tr>
<tr>
<td>1, iff x = 0</td>
</tr>
<tr>
<td>x * fac(x – 1), otherwise</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

| || Miranda         |
|-------------------|
| fac :: num -> num |
| fac x = 1, if x = 0 |
| fac x = x * fac (x – 1), otherwise |

<table>
<thead>
<tr>
<th>(* pascal *)</th>
</tr>
</thead>
<tbody>
<tr>
<td>function fac(n : integer) : integer;</td>
</tr>
<tr>
<td>var i, r: integer;</td>
</tr>
<tr>
<td>begin</td>
</tr>
<tr>
<td>r := 1;</td>
</tr>
<tr>
<td>for i := n downto 1 do</td>
</tr>
<tr>
<td>r := r * i;</td>
</tr>
<tr>
<td>fac := r;</td>
</tr>
<tr>
<td>end;</td>
</tr>
</tbody>
</table>
Type Inference

• The process of finding the type of an expression
  – If it fails, programmer has to type it explicitly
• Haskell uses it, PHP, C# too (a little bit)

{- Haskell: note type-inference -}
fac n = if n == 0 then 1 else n * fac (n-1)
{- or: -}
fac 0 = 1
fac n = n * fac ( n – 1 )

// C# type-inference
class A {
    static LinkedList<char> fill(int n) { .. }
    static void Main() {
        var X = fill(123);
    }
}
Lists

- Lists: most important data-structure in functional languages
  - Recursively defined/iterable
  - Contains zero, one, many items of the same type
- [1,4,5,9]
- [1 .. 10]
- ['a','b','c']
- []
- [[0,2,4,6],[1,3,5]]
List Comprehensions

- Define a list using a function for the item values

Miranda:

- \[x \mid x \leftarrow [1..10] ; x \mod 2 = 1\]  ||  \([1,3,5,7,9]\)
- Read ‘<-’ as ∈ for set-inclusion, ‘;’ as ‘and’

Haskell (mod is a function):

- \[i \mid i \leftarrow [1..10], ((\text{mod} i 2) == 1) \]
- Or force 'infix' function application
  - \[i \mid i \leftarrow [1..10], ((i `\text{mod}` 2) == 1) \]
  - \([0,5..20] \rightarrow \{0,5,10,15,20\}\)

Haskell note: functions/params should start with lower-case, types with upper case: “Func x” is wrong, “func x” is OK
Lists

• List access mostly recursive
  – Miranda: head/tail
    • hd[1,2,3] => [1]
    • tl[1,2,3] => [2,3]
  – Miranda: 'cons'
    • 1:[] => [1]
    • 1:[2,3] => [1,2,3]

• some languages have shortcut operations for efficiency: length(concat/reverse/sort
  – Provide an interface to an imperative library
Miranda:
length x = 0, if x = []
length x = 1 + length (tl x), otherwise

Or

length x = # x

{- Haskell -}
listLength x = if x==[] then 0 else 1+ListLength tail

{- or use the builtin length operator -}
listLength x = length(x)
Miranda
concatenate x y = y, if x = []
concatenate x y = (hd x) : concatenate (tl x) y, otherwise

{- Haskell -}
concatenate x y = if x == [] then y else ((head x) : concatenate (tail x) y)

{- or: -}
concatenate x y = x ++ y
Lists

• Can we build lists without using the builtin list type in a functional language?
  – No

• We lose efficiency by creating copies of lists, etc. BUT:
  – No hassle with
    • Next-element-ptr, previous-element-ptr
    • malloc/new, free/delete
  – Internally there's a garbage collector that frees unused lists/list-nodes
Polymorphic functions

- Function that accepts different actual argument types
  - Different from overloaded function which has N different instances for every of N different possible actual argument types

  || Miranda
  pair x y = [x, y]

  || .. can be used like:
  pair 1 2
  pair True False
  pair [1] [2]

  // C++: overloading
  list pair(int a, int b);
  list pair(bool a, bool b);
  list pair(list a, list b);

  // OR C++ polymorphic/template function:
  template<typename T>
  list<T> pair(T a, T b);
Polymorphic functions

• What is the type of a parameter in a polymorphic function?
  – Miranda: 'type variables'
    • Use this to enforce that which things in the parameter list should have equal types
    • Type inference can do this BUT error given late where now we can give error early: 'strong static checking'

|| Miranda
pair :: α -> (α -> [α]) || read as: (α,α)->[α], see 'currying' later on
pair x y = [x, y]

pair True 1 || error: detected as type(True) != type(1)
Higher Order Functions

- In functional languages, functions can be passed around
  - Function with a function argument = “higher order function”

|Miranda:|
tripple x  = x * 3
tripplelist x = map tripple x
tripplelist [1,2,3]  => [3,6,9]

|Haskell: -|
tripple x  = x * 3
tripplelist x = map tripple x
tripplelist [1,2,3]  {- [3,6,9] -}
Folding

- Foldr, foldl
  - Performs list-reduction (fold-right/left)

|| Miranda (and *exactly* the same for Haskell)
sum x y = x + y
sumlist x = foldr sum 0 x

|| sum(1 + sum(2 + sum(3 + 0)))  ==> 6
sumlist [1, 2, 3]

|| alternatively using the + operator
sumlist x = foldr (+) 0 x

foldl (/) 1 [1,2,3]  ==> 0.166666
|| ((1 / 1) / 2) / 3
foldr (/) 1 [1,2,3]  ==> 1.5
|| 3 / 2 / 1 / 1
• List reduction using boolean operators
  – 'fold' multiple values into one

{- Haskell: -}

all even (map (2*) [1..5])

any odd [ x^2 | x<-[1..5] ]
Currying

- Call a function with only some of its parameters
  - This delivers a partial function where the leftover parameters still need to be passed
  - “partial parametrization”
  - “currying”

Mult :: num -> (num -> num)
mult a b = a * b

tripple :: num -> num || remove first “num->” from mult
tripple = mult 3   || mult partially instantiated
Lazy Evaluation

- Applicative order reduction ("eager evaluation")
  - Evaluate inner expressions first, move outward
    - mult (mult 2 3) (mult 3 4)
    - mult 6 12
    - 72

- Normal order reduction ("lazy evaluation")
  - Evaluate outer expressions first, inner ones only when needed
    - mult (mult 2 3) (mult 3 4)
    - (mult 2 3) * (mult 3 4)
    - 6 * 12
    - 72
Lazy Evaluation

- May be very efficient (compared to eager evaluation)
  - foo(val, car(), boat())
    - Car() and boat() can both take a lot of compute power
    - Only the result of one is used, the other ignored

```c
// C: both 'car' and 'boat' are completely evaluated
int foo(int test, int a, int b) {
    if (test > 10)
        return a;
    else
        return b;
}
```

```haskell
{- Haskell: either car OR boat evaluated, not both. -}
foo test a b = if test > 10 then a else b
```
Lazy Evaluation

- A language with eager evaluation = strict semantics
  - Otherwise non-strict (of course)
- Allows us to manipulate infinite sized structures
  - Haskell/Miranda:
    - \([1..] \mid\mid\) list of all positive integers
    - head \([1..]\)
    - head (tail \([1..]\))
    - head ( tail ( map tripple \([1..]\)) )
Lazy-evaluation based I/O

- I/O is a list of request/responses to the operating system
  - I/O is thus a request/response *stream*
  - The I/O list has infinite length and grows with time
    - Only when done is actual I/O performed
Lazy Evaluation

- Most of the time **less** efficient than eager evaluation:

```
sumsum x = x + x
sumsum 4*5

|| Lazy
4*5 + 4*5
20 + 20
40

-----------------------------
2 mults + 2 adds

|| Eager
sumsum 20
20 + 20
40

-----------------------------
1 mult + 1 add

see “Graph Reduction”: lazy evaluation with bookkeeping*
Rebinding unbound variables

- In an expression some languages allow variables to be later specified
  - (instead of everything beforehand)

```
-- Haskell
zoo a = b * 100
  where
tmp = a * 2;
b = tmp * 2;
```

-- 'b' not declared here yet, only used!
Pattern Matching

- Given a set of patterns, search for the 1\textsuperscript{st} match -> execute its action
  - Pattern = <expression>, <condition>

|| Haskell/Miranda: Declare pattern Cond <bool> x y
cond True x y = x
cond False x y = y
  || Usage
cond (5 > 6) car boat

fac 0 = 1
fac (n + 1) = (n + 1) * fac n

f(n + 1) = .. allowed
f(n + m) = .. not allowed, ambiguous!

len [] = 0
will only match if non-empty
as there must be at least a head
len (a:b) = 1 + (len b)
uniq [] = []
uniq (a:(a:x)) = uniq (a:x)
uniq (a:x) = a:uniq x

unique x = uniq(sort(x))

**Miranda**
quicksort [] = []
quicksort (x:rest) = quicksort [a | a <- rest; a <= x] ++ x
++ quicksort [a | a <- rest; a > x]

**NOTE:** ++ is list concatenation
**NOTE:** [a | a <- rest; a > x] is list comprehension:
“return each element ‘a’ in rest, where a > x”

{- Haskell: -}
quicksort [] = []
quicksort (x:rest) = quicksort([a | a <- rest, a <= x]) ++
[x] ++
quicksort([a | a <- rest, a > x])
data Tree = Empty | Leaf [Char] | TreeNode Tree Tree

instance Show Tree where
    show Empty = "<>"
    show (Leaf x) = x
    show (TreeNode x y) = "<" ++ show x ++ show y ++ ">

zoo = do z <- [TreeNode (TreeNode (Leaf "aa") (Leaf "bb")) (Leaf "cc")]
    return z;
Monads / Monoids

• Idea: let a function return two typed values, instead of just one
  – One is the real user data
  – One is the meta data
    • For example:
      – Put error values
      – Put the global' variables you're used to in here

• Examples:
  – {int value, boolean infinite}
  – {int value, boolean not_a_number}
Monads

• To support tuples of \{\text{data, meta}\}, we need to
  – Have a type-system of such tuples
  – Ability to create \{\text{data}, \text{meta}\} given only \{\text{data}\}
  – Ability to select \{\text{data}\} or \{\text{meta}\} from \{\text{data}, \text{meta}\}

• Haskell:
  – Given monad \(M\) (meta data) and type \('t'\) (user data), create monad type \('M\ t'\)
  – Given value \('t'\) and monad \(M\), have a monad function \('t \rightarrow M\ t'\)
    • Called the 'unit function'
  – Operation \((M\ t) \rightarrow (t \rightarrow M\ u) \rightarrow M\ u\), which Haskell represents by the infix operator \(>>=\)
Monads

• Haskell:
  − Type def:
    • data Maybe t = Just t | Nothing
  − Unit function
    • return x = Just x
  − Binding:
    • (Just x) >>= f = f x
      − (just x) is obtained by applying 'f' defined as 'f x'
    • Nothing >>= f = Nothing
      − (Nothing) is obtained if by applying 'f' defined as Nothing
Monads

- Haskell monads have a simpler 'do' notation
  - identifier = 'do' (statement \n) expression \n  - 'do' thus allows statement sequencing

```
a = do x <- [1, 2, 3]
    return (Just x)

-- a results in [Just 1, Just 2, Just 3]
```

```
main = do putStrLn "hello"
        putStrLn "world"
        putStrLn ""
```
• Using monads we can (almost) write readable loops

module Main where

import Control.Monad.State

-- word separating tokens
whitespace = ['\n', ' ', '
', '	', '	', ' ']

--Type Counter = State Integer

--Returns the number of characters
wordCount :: String -> String

{--
doCount returns a Counter which is an instance of the Monad class. applying "0" to the Counter type initializes the Monad's state and generates the result of evaluating the string sent to doCount.
--}

doCount :: String -> Bool -> Counter Integer
doCount [] wasWs = do res <- getCount
  return res

doCount (x:xs) wasWs
  | not (elem x whitespace) && wasWs = do c <- incCount
  | not (elem x whitespace) && not wasWs = do c <- getCount
  | otherwise = do c <- getCount
    el <- doCount xs True
    return el

getCount :: Counter Integer
getCount = get

incCount :: Counter ()
incCount = modify (+1)

main = do interact wordCount
         putStrLn ""

---
Monads

- Monads to maintain semi-global state/variables
  - Keep the state around in parameters
  - Example: Random number generation

```haskell
-- problem: each call to randomNext requires a new 'rand'
-- otherwise we return the same value always!
randomNext:: Int -> Int
randomNext rand = newRand
  where
    newRand = 16807 * (12312312 `mod` 2836 * rand)
```
Example: Hamming Problem

- Give all numbers \(2^i \times 3^j \times 5^k\) in increasing order
  - '1' is a hamming number (i=j=k=0)
  - A hamming number multiplied by 2, 3, or 5 is also hamming
  - Can't make a hamming number any other way
Example: Hamming Problem

|| Miranda: multiply elts of list 's' by 'f'
mul f s = [f *x | x <- s]

|| return a list that starts with '1' followed by ..
ham = 1 : merge3 (mul 2 ham) (mul 3 ham) (mul 5 ham)

|| we multiplied the last hamming number by 2, 3, 5
|| now remove duplicates: Ex.: 10 = 2*5 AND 5*2
merge3 x y z = merge2 x (merge2 y z)

|| merge 2 sorted lists, removing duplicates as we go
merge2 (x:xs) (y:ys)
  = (x:merge2 xs (y:ys)), if x < y
  = (x:merge2 xs ys), if x = y
  = (y:merge2 (x:xs) ys), if x > y
Example: Lisp

• Properties
  – Case insensitive
  – Recursion + higher-order functions
    • No type-system
  – Automatic memory management
  – Hard to read for mere humans (but easy for a compiler)
  – Prefix notation of expressions: “* 3 4” ISO “3 * 4”

;;Std/old-style Lisp
(define (fac x)
  (cond ((= x 0) 1) (t (* x (fac (- x 1))))))

;;Common Lisp:
(defun factorial(x)
  (if (< x 2) 1 (* x (factorial (- x 1)))))
Example: Common Lisp

- (eval (/ (/ 1 34) 3))
  - (1 / 34) / 3
  - Prints “1/102”, it thus keeps fractions internally instead of floating point numbers!

- (eval (float (/ 1 3)))
  - Explicit cast from fraction to floating point

- I/O
  - (read), (write “hi”)

- ` vs not `
  - ` means pass the symbol, not ` means evaluate symbol
Example: Common Lisp

; create a two dimensional array
(setq a1 (make-array `(3,4)))

; get array element at (0, 0)
(aref a1 0 0)

; set array element (0, 0) to the symbol abcdef
(setf (aref a1 0 0) (`abcdef))

; set array element (0, 0) to the string “hi”
(setf (aref a1 0 0) "hi")

Note:
- setq assigns to symbol
  C: int a = ...
- setf assigns a value to a place
  C: *a = ...
Example: Common Lisp

• Symbol properties:
  - (get `foo `color)
    • Get foo's color
  - (setf (get `foo `color) `red)
    • Set foo's color to RED
  - Problem: there's only one foo over the whole program, naming conflicts occur easily!

• Structures / Records
  - (defstruct struct1 color size shape)
  - (setq obj1 (make-struct1 :size 'small :color 'green :shape 'round)
Example: Common Lisp

- Hashtables
  - `(setq ht (make-hash-table))
    • Create hashtable and assign a pointer to ht
  - `(gethash `a ht)
    • Get the value associated with key 'a'
  - `(setf (gethash `a ht) `b)
    • Assign symbol 'b' to hash-entry associated with 'a'

- Lists
  - `(first '(2 4 8)) ; 2
  - `(rest '(2 4 8)) ; (4, 8)
  - `(cons 1 (cons 2 `(3 4))) ; (1,2,3,4)
  - `(member 2 `(4, 3, 7, 2, 1, 6)) ; (2, 1, 6)
Example: OCaML

- **ML = Meta Language**
  - Functional
  - Pattern matching
  - Polymorphism

- **Ocaml = either**
  - Objective (Categorically Abstract Machine Language)
  - Objective (Concurrent Abstract Meta Language)
  - Adds object-orientation to ML

- **F# by Microsoft is Ocaml for DotNet platform..**
OCaml

- let <ident> <params*> = <expression> ;;
  - Defines a function with name 'ident'
    - Params are variables that can be used in 'expression'
      - () not used in calls/parameter declarations
    - Automatic type-inference is used to find the types of both expression and params
      - Uses different operators for floating point and integer operations
        - <op> = integer operation
        - <op>. = floating point operation

```
# let y a b c = a * b * c;;
val y : int -> int -> int -> int = <fun>
# let z a b c = a * . b * . c;;
val z : float -> float -> float -> float = <fun>
# y 1 2 3;;
- : int = 6
# z 1.0 2.0 3.0 ;;
- : float = 6.
```
OCaml

- Recursive functions need to be marked as such:
  - let x a b c = ..
  - let rec x a b c

- Datatypes = \{int, float, bool, char, string, tuple, array, list, records, variants, objects\}
OCaml

- We can create partial functions (currying)
- We can pass functions around (higher order functions)

```ocaml
# let deriv f dx = function x -> (f(x +. dx) -. f(x)) /. dx;;
val deriv : (float -> float) -> float -> float -> float = <fun>

# let sin' = deriv sin 1e-6;;
val sin' : float -> float = <fun>

# sin' pi;;
- : float = -0.999998732663419765

# let compose f g = function x -> f(g(x));;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>

# let square x = x *. x;;
val square : float -> float = <fun>

# let cos2 = compose square cos;;
val cos2 : float -> float = <fun>

# let deriv f dx = function x -> (f(x +. dx) -. f(x)) /. dx;;
val deriv : (float -> float) -> float -> float -> float -> float = <fun>

# let sin' = deriv sin 1e-6;;
val sin' : float -> float = <fun>

# sin' pi;;
- : float = -0.999998732663419765
```
OCaml

- Lists are built using the empty list and appending items
  - Empty list = []
  - Append operator = '::'
  - List entry separator = ';;'

- Note: can't append, only prepend, list must be uniformly typed

```ocaml
# let l = ["a"; "b"];;
val l : string list = ["a"; "b"]
# let k = l :: ["c"] ;;
This expression has type string but is here used with type string list
# let k = l :: "c";;
This expression has type string but is here used with type string list list
# let k = "c" :: l ;;
val k : string list = ["c"; "a"; "b"]
```
OCaml

- Data structure manipulation via pattern matching
  - A match statement is like a case statement
  - Note polymorphism: sort works on strings-list and integer list!

```ocaml
# let rec insert elt lst =
  match lst with
  | []         -> [elt]
  | head :: tail -> if elt <= head then elt :: lst
                   else head :: insert elt tail;;
val insert : 'a -> 'a list -> 'a list = <fun>
# let rec insertionsort lst =
  match lst with
  | []         -> []
  | head :: tail -> insert head (insertionsort tail);;
val sort : 'a list -> 'a list = <fun>
# let l = ['is'; 'a'; 'tale'; 'told'; 'etc.'];;
val l : string list = ['is'; 'a'; 'tale'; 'told'; 'etc.'];
# sort(l);;
- : string list = ['a'; 'etc.'; 'is'; 'tale'; 'told']
```

```ocaml
# let rec map f l = match l with
  | [] -> []
  | hd::tl -> f hd :: map f tl ;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# let l = [1; 2; 3] ;;
val l : int list = [1; 2; 3]
# let double x = x * 2;;
val double : int -> int = <fun>
# map double l  ;;
- : int list = [2; 4; 6]
```
OCaml

- User defined records and variants (aka enumeration type)

# type complex = {rat:float; irat:float};;
type complex = { rat : float; irat : float; }
-- ocaml finds return type by looking at 'rat' and irat fields
# let complexmult a b = {rat = a.rat *. b.rat;
  irat = a.irat *. b.irat } ;;
val complexmult : complex -> complex -> complex = <fun>
# complexmult {rat=1.0; irat=2.0} {rat=3.0; irat=4.0};;
- : complex = {rat = 3.; irat = 8.}

# type sign = Positive | Negative;;
type sign = Positive | Negative
# let sign_int n = if n >= 0 then Positive else Negative;;
val sign_int : int -> sign = <fun>
# sign_int 3;;
- : sign = Positive
OCaml

- Ocaml also has imperative programming primitives
  - In 'pure' functional programs, you can't express for/while as the loop iterator is of the form
    - $X = X + 1$
    - *Which is not a correct math formula!*
  - However: always using recursion = slow!!

```
# let add_vect v1 v2 =
  let len = min (Array.length v1) (Array.length v2) in
  let res = Array.create len 0.0 in
  for i = 0 to len - 1 do
    res.(i) <- v1.(i) +. v2.(i)
  done;
  res;;

val add_vect : float array -> float array -> float array = <fun>
```

```
# add_vect [| 1.0; 2.0 |] [| 3.0; 4.0 |];;
- : float array = [|4.; 6.|]
```
OCaml

• Modules are used to package functions / records / types, syntax =
  • module <ident> = struct <decls> end;;
  – Modules are named and accessed over their name
• Modules have a separate specification part
  • Module type <ident> = sig .. end ;;
  – Abstract data type (ADT) support by not specifying module internal types
  • Only their name is mentioned

module mylist = struct
  type list = Empty | list * node
  let empty = Empty
  let myAdd l n = ...
end;;

module type mylist = sig
  type list;
  let empty: list
  let myAdd : list -> int -> list
end;;
Parameterized types are via ADT usage

- Functors: a function mapping one data structure to another

- Generic X<A> is thus a functor X(A)
  - Java's list<A> is thus a functor list(A), with 'list' a function

```ocaml
#type comparison = Less | Equal | Greater;;
#module type ORDERED_TYPE =
  sig
    type t
    val compare: t -> t -> comparison
  end;;

#module Set =
  functor (Elt: ORDERED_TYPE) ->
    struct
      type element = Elt.t
      type set = element list
      let empty = []
      let rec add x s =
        match s with
        [] -> [x]
        | hd::tl ->
          match Elt.compare x hd with
          Equal -> s
          | Less   -> x :: s
          | Greater -> hd :: add x tl
      let rec member x s =
        match s with
        [] -> false
        | hd::tl ->
          match Elt.compare x hd with
          Equal -> true
          | Less   -> false
          | Greater -> member x tl
    end;;
```
OCaml

- Classes are structures with functions
  - With 'new' we can instantiate an object
  - With the '#' operator we access objects
    - This is the '.' is Java or the '->' in C++

```ocaml
# class point =
  object
    val mutable x = 0  -- declared mutable so its value can change
                       -- otherwise this would be a 'final' or 'const'

    method get_x = x
    method move d = x <- x + d
  end;;

# let p = new point;;
val p : point = <obj>
```
OCaml

• Ocaml does not have classic constructors
  – Use a function that returns an initialized object
  – self/this must be given a name to access it

```
# class point = fun x_init ->
  object
    val mutable x = x_init
    method get_x = x
    method move d = x <- x + d
  end;;

-- a point is a function that needs an int to
-- create a real point
# class printable_point x_init =
  object (s)
    -- name this 's'
    val mutable x = x_init
    method get_x = x
    method move d = x <- x + d
    -- here we access 'this' via 's'
    method print = print_int s#get_x
  end;;
```

```
# new point;;
- : int -> point = <fun>

# (new point 3)#get_x;;
- : int = 3
```
OCaml

- Multiple inheritance model
  - Multiple method/field definitions: discard all but last
- Abstract methods must be marked 'virtual'
  - the class must be marked virtual too
  - Like Java's abstract methods and classes

class virtual abstract_point x_init =
object (self)
  method virtual get_x : int
  method get_offset = self#get_x - x_init
  method virtual move : int -> unit
end;;

class point x_init =
object
  inherit abstract_point x_init
  val mutable x = x_init
  method get_x = x
  method move d = x <- x + d
end;;
- Optionally functional objects
  - Whenever an instance variable is changed: create a copy of the object
    - Write method body using \(< \ldots >\)
      - Instead of {} as normal
    - Very slow, BUT we gain back functional-transparency!
  - When inheriting: a 'copying' method stays copying in subclasses

```ocaml
class functional_point y =
  object
    val x = y
    method get_x = x
    method move d = \(< x = x + d >\)
  end;;
```

Move returns a copy of 'this'
Next week: logic programming