Programming Languages 2012: Functional Programming: ML

(Based on [Sethi 1996] and [Leroy *et al.* 2012; OCaml])

Yih-Kuen Tsay

1 Functions on Lists

Lists

- Lists are the original data structure of functional programming, just as arrays are that of imperative programming.
- A list in ML is a sequence of zero or more elements of the same type, enclosed by a pair of brackets [and] and separated by ;. So, $[1; 2; 3]$ is a list of integers.
- [ ] denotes the empty list.
- Structure:
  - A list is either empty (i.e., equals [ ]),
  - or it has the form $a :: y$, where element a is the head of the list, and the sublist y is the tail of the list.
  - For example, $[1; 2; 3] \equiv 1 :: [2; 3] \equiv 1 :: 2 :: [3] \equiv 1 :: 2 :: 3 :: [\,]$.

Operations on Lists

- OCaml provides the following basic functions (operations) on lists:

Function	Description
$=$	equality test, particularly with [ ]
$::$	infix list constructor (read "cons")
List.hd	return the head
List.tl	return the tail

- OCaml also provides the following functions (which could have been left for the user to define):

Function	Description
@	append/concatenate two lists
List.rev	reverse the list
List.length	count the number of elements
List.nth	return the nth element

User-Defined Functions on Lists

- Most functions on lists consider the elements of a list one by one and behave as follows:

  let rec $f\ x =$
  if "list x is empty" **then** ...
  else "something involving head/tail of x and f"

- A function like f is said to be *linear recursive* if f appears only once on the right side of =. For example,

  let rec *length* $x =$ **if** $x = [\,]$ **then** 0
  else $1 +$ *length* (*List.tl* x)

Precedence of Operations

The usual levels of precedence (from high to low):

function application
`**`
`* / *. /. mod`
`+ - +. -.`
`::`
`@ ^`
`< <= = != <> >= >`

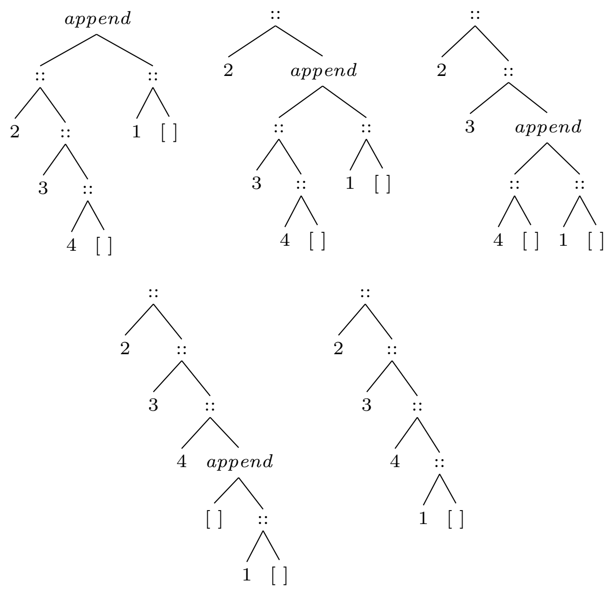
Append

We may define a function that behaves the same way as @.

let rec *append* $x\ z =$
if $x = [\,]$ **then** z
else *List.hd* x :: *append* (*List.tl* x) z

append $[2; 3; 4]\ [1] \equiv [2; 3; 4; 1]$

Append in Action



Reverse

We may also define a function that behaves the same way as *List.rev*.

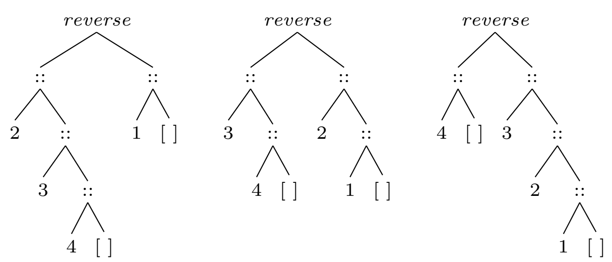
let rec *reverse* x z =
  if $x = [\,]$ **then** z
  else *reverse* (*List.tl* x) (*List.hd* $x :: z$)

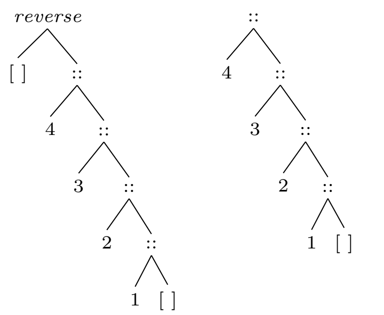
reverse $[2;3;4]$ $[1] \equiv [4;3;2;1]$

let *rev* x = *reverse* x $[\,]$

  rev $[1;2;3;4] \equiv [4;3;2;1]$

Reverse in Action





2 Pattern Matching

Patterns and Cases

- Observe that

$$\begin{array}{rcl} length\ [\,] & \equiv & 0 \\ length\ (a :: y) & \equiv & 1 + length\ y \end{array}$$

- We may define *length* according to the patterns of the input as follows.

  let rec *length* x =
    match x **with**
      $[\,] \rightarrow 0$
    $|$ $a :: y \rightarrow 1 + length\ y$

- Alternatively,

  let rec *length* = **function**
      $[\,] \rightarrow 0$
    $|$ $a :: y \rightarrow 1 + length\ y$

  This construct of **function** permits exactly one formal parameter.

Patterns and Cases (cont.)

- Similarly,

  let rec *append* x z =
    match x **with**
      $[\,] \rightarrow z$
    $|$ $a :: y \rightarrow a :: append\ y\ z$

  let rec *reverse* x z =
    match x **with**
      $[\,] \rightarrow z$
    $|$ $a :: y \rightarrow reverse\ y\ (a :: z)$

- Patterns on tuples can be expressed more compactly.

  let *first* $(x, y) = x$

  let *second* $(x, y) = y$

Patterns and Cases (cont.)

- As we have seen, patterns and cases lead to more readable code.
- An underscore _ denotes a "don't-care" pattern.

  let *first* $(x, \_) = x$

- The same formal parameter may not be used more than once in a pattern. So, the pair $(a, a :: y)$ is not a legal pattern.

3 Map and Reduce: Functions as First-Class Values

Applying Functions Across List Elements

- A *filter* is a function that copies a list, making useful changes to the elements as they are copied.
- The simplest one is *copy*:

```
# let rec copy x =
    match x with
      [] -> []
    | a::y -> a::(copy y);;
val copy : 'a list -> 'a list = <fun>
```

Applying Functions Across List Elements (cont.)

- Below is a filter function for squaring each list element:

```
# let square n = n * n;;
val square : int -> int = <fun>

# let rec copysq x =
    match x with
      [] -> []
    | a::y -> square a :: copysq y;;
val copysq : int list -> int list = <fun>
```

- We will study a function called *map*, which is a tool for building a filter out of an input function.

Accumulate a Result

- Below is a function for computing the sum of a list of integers:

```
# let rec sum_all = function
      [] -> 0
    | a::y -> a + sum_all y;;
val sum_all : int list -> int = <fun>
```

- And, below is a function for computing the product of a list of integers:

```
# let rec product_all = function
      [] -> 1
    | a::y -> a * product_all y;;
val product_all : int list -> int = <fun>
```

- We will study a function called *reduce*, which is a generalization of such accumulation functions.

Map and Reduce

- Below are the very useful *map* and *reduce*:

  let rec *map f x* =
    match *x* **with**
      $[\,] \to [\,]$
    $|\ a :: y \to (f\ a) :: map\ f\ y$

  let rec *reduce f x v* =
    match *x* **with**
      $[\,] \to v$
    $|\ a :: y \to f\ a\ (reduce\ f\ y\ v)$

- Both functions are "higher-order" functions, as they take another function as an input.
- They are supported in OCaml as `List.map` and `List.fold_right`.

The Utility of Map

- Suppose we have now defined *map*:

```
# let rec map f x =
    match x with
      [] -> []
    | a::y -> (f a) :: (map f y);;
val map : ('a -> 'b) -> 'a list ->
'b list = <fun>
```

- And, also the following functions:

```
# let square n = n * n;;
val square : int -> int = <fun>
# let first (x,y) = x;;
val first : 'a * 'b -> 'a = <fun>
# let second (x,y) = y;;
val second : 'a * 'b -> 'b = <fun>
```

The Utility of Map (cont.)

- Using *map* to apply a function to each list element:

  ```
  # map square [1; 2; 3];;
  - : int list = [1; 4; 9]
  # map first [(1,"a"); (2,"b"); (3,"c")];;
  - : int list = [1; 2; 3]
  # map second [(1,"a"); (2,"b"); (3,"c")];;
  - : string list = ["a"; "b"; "c"]
  ```

- In OCaml, `List.map` may be used instead.

The Utility of Reducton

```
# let rec reduce f x v =
    match x with
      [] -> v
    | a::y -> f a (reduce f y v);;
val reduce : ('a -> 'b -> 'b) -> 'a list
-> 'b -> 'b = <fun>
# let add x n = String.length x + n;;
val add : string -> int -> int = <fun>
# let mult x n = String.length x * n;;
val mult : string -> int -> int = <fun>
# reduce add ["1"; "23"; "456"] 0;;
- : int = 6
# reduce mult ["1"; "23"; "456"] 1;;
- : int = 6
```

In OCaml, `List.fold_right` may be used instead.

Anonymous Functions

An *anonymous function*, a function without a name, has the form

$$\textbf{fun}\ \langle \textit{formal-parameter} \rangle \rightarrow \langle \textit{body} \rangle$$

Examples:

```
# fun x n -> String.length x + n;;
- : string -> int -> int = <fun>

# reduce (fun x n -> String.length x + n)
         ["1"; "23"; "456"] 0;;
- : int = 6
```

4 Type Inference

Type Inference

Wherever possible, ML infers types without help from the user.

```
# 3.0 * 4;;
Characters 0-3:
  3.0 * 4;;
  ^^^
Error: This expression has type float but
an expression was expected of type int
# 3.0 *. 4;;
Characters 7-8:
  3.0 *. 4;;
         ^
Error: This expression has type int but
an expression was expected of type float
# 3.0 *. 4.0;;
- : float = 12.
```

Type Inference (cont.)

```
# let add x y = x + y;;
val add : int -> int -> int = <fun>

# let add x y = x +. y;;
val add : float -> float -> float = <fun>
```

Parametric Polymorphism

- A definition of the *identity* function:

  ```
  # let id x = x;;
  val id : 'a -> 'a = <fun>
  ```

- The leading quote in `'a` identifies it as a type parameter.

- A *polymorphic* function can be applied to arguments of more than one type.

- *Parametric polymorphism* is a special kind of polymorphism in which type expressions are parameterized.

Parametric Polymorphism (cont.)

```
# [1; 2; 3];;
- : int list = [1; 2; 3]
# ["one"; "two"; "three"];;
- : string list = ["one"; "two"; "three"]
```

```
# let rec len = function
      [] -> 0
    | a::y -> 1 + len y;;
val len : 'a list -> int = <fun>

# len ["one"; "two"; "three"];;
- : int = 3
# len [1; 2; 3];;
- : int = 3
```

Parametric Polymorphism and Type Inference

```
# let rec sum x =
    match x with
      [] -> 0
    | a::y -> a + sum y;;
val sum : int list -> int = <fun>

# let rec sum = function
      [] -> 0.
    | a::y -> a +. sum y;;
val sum : float list -> float = <fun>
```

5 Types

Types

- *Type* declarations define types corresponding to data structures.
- Value Constructors

  ```
  # type direction = North | South | East | West;;
  type direction = North | South | East | West
  ```

  This declaration introduces a basic type `direction`; the associated set of values is `{North, South, East, West}`.

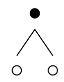
- Parameterized Value Constructors

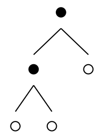
  ```
  # type bitree = Leaf | Node of bitree*bitree;;
  type bitree = Leaf | Node of bitree * bitree
  ```

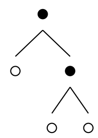
  A value of type `bitree` is either the constant `Leaf` or it is constructed by applying `Node` to a pair of values of type `bitree`.

Types (cont.)

Leaf


Node (Leaf, Leaf)


Node (Node (Leaf, Leaf), Leaf)


Node (Leaf, Node (Leaf, Leaf))

Operations on Constructed Values

```
# let rec leafcount = function
      Leaf -> 1
    | Node (l,r) -> leafcount l + leafcount r;;
val leafcount : bitree -> int = <fun>
# leafcount (Node (Node (Leaf, Leaf), Leaf));;
- : int = 3

# let isleaf = function
      Leaf -> true
    | Node _ -> false;;
val isleaf : bitree -> bool = <fun>
```

Operations on Constructed Values (cont.)

```
# let left = function
    Node (l,r) -> l;;
Characters 11-39:
  ..........function
      Node (l,r) -> l..
Warning 8: this pattern-matching is not
exhaustive. Here is an example of a value
that is not matched:
Leaf
val left : bitree -> bitree = <fun>

# let right = function
    Node (l,r) -> r;;
```

Operations on Constructed Values (cont.)

```
# let rec leafcount x =
    if isleaf x then 1
   else leafcount (left x) + leafcount (right x);;
```

```
val leafcount : bitree -> int = <fun>

# leafcount (Node (Node (Leaf, Leaf), Leaf));;
- : int = 3
```

A Differentiation Function

let rec $d\ x\ E =$
  if "E is a constant" **then** 0
  else if "E is the variable x" **then** 1
  else if "E is another variable" **then** 0
  else if "E is the sum $E_1 + E_2$ "
    then $d\ x\ E_1 + d\ x\ E_2$
  else if "E is the product $E_1 * E_2$ "
    then $(d\ x\ E_1) * E_2 + E_1 * (d\ x\ E_2)$

A Differentiation Function (cont.)

```
type expr =
     Constant of int
   | Variable of string
   | Sum of expr*expr
   | Product of expr*expr
let zero = Constant 0
let one = Constant 1
let u = Variable "u"
let v = Variable "v"
```

"$(u + v) * 1$" is represented as "`Product (Sum (u,v), one)`".

A Differentiation Function (cont.)

```
# let rec d x f =
    match x, f with
      _, Constant _ -> zero
    | Variable s, Variable t ->
        if s=t then one else zero
   | x, Sum (e1,e2) -> Sum ((d x e1),(d x e2))
    | x, Product (e1,e2) ->
      let term1 = Product ((d x e1),e2) in
      let term2 = Product (e1,(d x e2)) in
      Sum (term1,term2);;
```

Polymorphic Types

```
# type 'a nulist = Nil | Cons of 'a * ('a nulist);;
type 'a nulist = Nil | Cons of 'a * 'a nulist

# Nil;;
- : 'a nulist = Nil
# Cons (1, Cons (2, Nil));;
- : int nulist = Cons (1, Cons (2, Nil))
# Cons ("1", Cons ("2", Nil));;
- : string nulist = Cons ("1", Cons ("2", Nil))
```

6 Exceptions

Exceptions

Exceptions are a mechanism for handling special cases or failures that occur during the execution of a program.

```
# List.hd [];;
Exception: Failure "hd".

# exception Nomatch;;
exception Nomatch

# let rec member a x =
    if x=[] then raise Nomatch
    else if a = List.hd x then x
    else member a (List.tl x);;
val member : 'a -> 'a list -> 'a list = <fun>

# member 3 [1;2;3;1;2;3];;
- : int list = [3; 1; 2; 3]
# member 4 [1;2;3;1;2;3];;
Exception: Nomatch.
```

Exceptions with Arguments

Exceptions may be attached with one or more values.

```
# exception Nomatch of string;;
exception Nomatch of string

# let rec member a x =
    if x=[] then raise (Nomatch "member")
    else if a = List.hd x then x
    else member a (List.tl x);;
val member : 'a -> 'a list -> 'a list = <fun>
# member 4 [1;2;3;1;2;3];;
Exception: Nomatch "member".
```

Exception Handling

Exceptions can be caught or handled by using the following syntax:

try $\langle expr\rangle_1$ **with** $\langle exception\text{-}name\rangle \rightarrow \langle expr\rangle_2$

```
# exception Oops;;
exception Oops
# exception Other;;
exception Other

# try (raise Oops) with Oops -> 0;;
- : int = 0

# try (raise Other) with Oops -> 0;;
Exception: Other.
```

Finding Exception Handlers

Exceptions are handled dynamically.

If f calls g, g calls h, and h raises an exception, then we look for handlers along the call chain h, g, f. The first handler along the chain catches the exception.

```
# exception Neg;;
exception Neg
# let s m n =
    if m >= n then m - n
    else raise Neg;;
val s : int -> int -> int = <fun>

# s 5 10;;
Exception: Neg.
```

Finding Exception Handlers (cont.)

```
# let subtract m n =
    try (s m n)
    with Neg -> 0;;
val subtract : int -> int -> int = <fun>

# subtract 5 10;;
- : int = 0
```

7 Little Quilt in ML

Little Quilt in ML

```
type texture = WTriangle | BTriangle
type direction = NE | SE | SW | NW

type square = texture * direction
type row = square list
type quilt = row list

let sqa = (WTriangle,NE)
let sqb = (BTriangle,NE)
let a = [[sqa]]
let b = [[sqb]]
```

Little Quilt in ML (cont.)

```
exception Failed

let rec sew q1 q2 =
  match q1, q2 with
    [], [] -> []
  | l::x, r::y -> (l @ r) :: (sew x y)
  | _, _ -> raise Failed
```

The *sew* Operation in Action

```
[[(WTriangle,NE);(WTriangle,SW)];
 [(BTriangle,SW);(WTriangle,NE)]]
```

```
[[(WTriangle,SE);(WTriangle,NW)];
 [(WTriangle,NW);(BTriangle,SE)]]
```

sew

↓

```
[[(WTriangle,NE);(WTriangle,SW);(WTriangle,SE);(WTriangle,NW)];
 [(BTriangle,SW);(WTriangle,NE);(WTriangle,NW);(BTriangle,SE)]]
```

Little Quilt in ML (cont.)

```
let clockwise = function
    NE -> SE
  | SE -> SW
  | SW -> NW
  | NW -> NE

let turnsq = function
    (tex,dir) -> (tex, clockwise dir)
```

Little Quilt in ML (cont.)

```
let compose f g = fun x -> f (g x)

let rec emptyquilt = function
    [] -> true
  | []::tl -> emptyquilt tl
  | _ -> false

let rec turn q =
  if emptyquilt q then []
  else (List.rev
       (List.map (compose turnsq List.hd) q))
       ::
       (turn (List.map List.tl q))
```

The *turn* Operation in Action

```
x =
[[(WTriangle,NE);(WTriangle,NE);(WTriangle,NE)];
 [(BTriangle,NE);(WTriangle,NE);(WTriangle,NE)];
 [(BTriangle,NE);(BTriangle,NE);(WTriangle,NE)]]
```

```
List.map List.hd x =
[(WTriangle,NE);
 (BTriangle,NE);
 (BTriangle,NE)]
```

```
List.map (compose turnsq List.hd) x =
[(WTriangle,SE);
 (BTriangle,SE);
 (BTriangle,SE)]
```

```
List.rev (List.map (compose turnsq List.hd) x) =
[(BTriangle,SE);(BTriangle,SE);(WTriangle,SE)]
```

Little Quilt in ML (cont.)

```
let unturn q = turn (turn (turn q))

let pile q1 q2 =
  unturn (sew (turn q2) (turn q1))
```

Little Quilt in ML (cont.)

The `unturn` function could be made more efficient with the following auxiliary functions.

```
let counterclockwise = function
    NE -> NW
  | SE -> NE
  | SW -> SE
  | NW -> SW

let unturnsq = function
  (tex,dir) -> (tex, counterclockwise dir)
```

Displaying a Quilt

```
let encode = function
    (WTriangle,NE) -> "◹"
  | (WTriangle,SE) -> "◿"
  | (WTriangle,SW) -> "◺"
  | (WTriangle,NW) -> "◸"
  | (BTriangle,NE) -> "◥"
  | (BTriangle,SE) -> "◢"
  | (BTriangle,SW) -> "◣"
  | (BTriangle,NW) -> "◤"
```

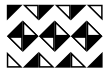
Displaying a Quilt (cont.)

```
let cat r = List.fold_right (^) r ""

let showrow r =
  let encodings = List.map encode r in
  print_endline (cat encodings)

let show q = List.map showrow q
```

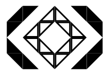
Example Quilt One



```
let slice =
  let aa = pile a (turn (turn a)) in
  let bb = pile (unturn b) (turn b) in
  let p = sew bb aa in
  let q = sew aa bb in
  pile p q

let quilt1 =
  let q = sew slice slice in
  sew q slice
```

Example Quilt Two



```
let quilt2 =
  let bb = pile (turn b) (unturn b) in
  let ba = pile (unturn b) (turn a) in
  let c_nw = sew bb ba in
  let c_ne = turn c_nw in
  let c_se = turn c_ne in
  let c_sw = turn c_se in
  let p = pile (turn a) (unturn a) in
  let q = pile (turn (turn a)) a in
  let top = sew (sew c_nw p) (sew q c_ne) in
  let bot = sew (sew c_sw q) (sew p c_se) in
  pile top bot
```

8 Some Imperative Constructs

Arrays

```
# [|1;2;3|];;
- : int array = [|1; 2; 3|]

# Array.make 10 0;;
- : int array = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]
```

```
# let a = [|1;2;3|];;
val a : int array = [|1; 2; 3|]

# Array.get a 1;;
- : int = 2
# a.(1);;
- : int = 2
```

Arrays (cont.)

```
# let a = [|1;2;3|];;
val a : int array = [|1; 2; 3|]

# Array.set a 1 4;;
- : unit = ()
# a;;
- : int array = [|1; 4; 3|]

# a.(2) <- 5;;
- : unit = ()
# a;;
- : int array = [|1; 4; 5|]
```

References

```
# let i = ref 0;;
val i : int ref = {contents = 0}

# i;;
- : int ref = {contents = 0}
# !i;;
- : int = 0
# i := 1;;
- : unit = ()
# !i;;
- : int = 1
# i := !i + 1;;
- : unit = ()
# !i;;
- : int = 2
```

The While-Do Statement

```
# let a = Array.make 10 0;;
val a : int array = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]

# let i = ref 0;;
val i : int ref = {contents = 0}

# while !i <= 9 do
   (a.(!i) <- !i; i := !i + 1)
  done;;
- : unit = ()

# a;;
- : int array = [|0; 1; 2; 3; 4; 5; 6; 7; 8; 9|]
```