March 18, 2002
6898: Advanced Topics in Software Design
MIT Lab for Computer Science
Daniel Jackson

Objective Caml
topics for today

• datatypes (variants)
• side-effects and the unit type
• inferred types and parametric polymorphism
• new notions (from ML)

• let bindings, functions, closures, lists
• familiar notions (from Scheme)
functions

applying an anonymous function

```ml
# (fun x -> 2 * x) 3;; - : int = 6
```

declaring a function and applying it

```ml
# let dbl = fun x -> 2 * x;; val dbl : int -> int = <fun>
# dbl 3;; - : int = 6
```

functionals, or higher-order functions

```ml
# let twice = fun f -> (fun x -> (f (f x)));; val twice : ('a -> 'a) -> 'a -> 'a = <fun>
# (twice dbl) 3;; - : int = 12
```

applying an anonymous function

```ml
# (fun x -> 2 * x) 3;; - : int = 6
```
let bindings

a let expression binds a variable to a value

```
# let x = 3 and y = 4 in x + y;;
```

read-eval-print-loop uses let instead of define

```
# let x = 5;; val x : int = 5
# x;;
```

recursive let

```
# let rec fact i = if i = 0 then 1 else i * fact (i - 1);;
```

a let expression binds a variable to a value

```
# let x = 5;; val x : int = 5
# let x = 5;;
```

Let bindings
Let is lexical

\[ \begin{align*}
\text{let } k = 5 ; & \quad \text{val } k : \text{int} = 5 \\
\text{let } f = \text{fun } x \to x + k ; & \quad \text{val } f : \text{int} \to \text{int} = \text{<fun>} \\
\text{f } 3 ; & \quad - : \text{int} = 8 \\
\text{let } k = 6 ; & \quad \text{val } k : \text{int} = 6 \\
\text{let } f = \text{6} ; & \quad \text{val } f = 6 \\
\text{let } f = \text{8} ; & \quad - : \text{int} = 8 \\
\text{let } f = \text{3} ; & \quad \text{val } f = \text{<fun>} \\
\text{let } f = \text{int } \to \text{int} = \text{fun } x \to x + k ; & \quad \text{val } f : \text{int} \to \text{int} = \text{<fun>} \\
\text{let } k = 5 ; & \quad \text{val } k = 5 \\
\text{let } k = 5 ; & \quad \text{val } k = 5
\end{align*} \]

let vs. define

no side-effecting top-level define built-in
tuples

<fun> = -
  - string: unit
    - print_string
      #
      () = unit
        -
          ();
            ()
                #
                empty tuple, used instead of `void`
        
    - : int = 2
      -
        "and x;"
          #
          
    - : int = 1
      -
        "let x;"
          #
          
val x : int * int = 1, 2

# let x = 1, 2;;

tuple constructor
function arguments

tupled form: like in an imperative language

```
let diff (i,j) = if i < j then j-i else i-j;;
val diff : int * int -> int = <fun>
```

```
(diff 3 4);;  
- : int = 1
```

curried form: stages the computation

```
let diff i j = if i < j then j-i else i-j;;
val diff : int -> int -> int = <fun>
```

```
(diff 3) 4;;
- : int = 1
```

This function is applied to too many arguments

```
let diff (i,j) = if i > j then j-i else i-j;;
```

```
<fun>
```

```
val diff : int * int -> int = <fun>
```

```
let diff (i,j) = if i > j then j-i else i-j;;
```

```
<fun>
```

tupled form: like in an imperative language

```
function arguments
```
[] = -
  "[]" #
empty list is polymorphic

This expression has type 'a list but is here used with type int

```
[[2]] #
[[1]] = -
  "[[1]]" #
```

Lists are homogeneous

```
[1; 2] = -
  "[1; 2]" #
```

```
[1; 2] = -
  "[1; 2]" #
```

Lists
polymorphic functions

The simplest polymorphic function

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

A polymorphic function over lists

```
# let cons e l = e :: l;;val cons : 'a -> 'a list -> 'a list = <fun>
# cons 1 2;;
- : int list = [1]
```

This expression has type int but is here used with type 'a list

```
let cons e l = e :: l;;
```

A polymorphic function over lists

```
<fun> = <fun>;-
```

The simplest polymorphic function
datatypes

```ocaml
# type color = Red | Green | Blue;;
# type color = Red | Green | Blue
# Red;;
- : color = Red
# [Red ; Green];;
- : color list = [Red; Green]
```

constructors can take arguments

```ocaml
# type numtree = Empty | Tree of numtree * int * numtree;;
# type numtree = Empty | Tree of numtree * int * numtree
# Empty;;
- : numtree = Empty
# Tree (Empty, 3, Empty);;
- : numtree = Tree (Empty, 3, Empty)
```

```ocaml
[50x63]10
datatypes
```

```ocaml
# color list = [Red, Green];;
# [Red ; Green];;
# color = Red
# Red;;
- : color = Red
# type color = Red | Green | Blue
# type color = Red | Green | Blue
```

a simple datatpe
patterns

a function on number trees

```
# type numtree = Empty | Tree of numtree * int * numtree;;
# let rec treesum t =   match t with Empty -> 0   | Tree (t1, i, t2) -> i + treesum t1 + treesum t2;;
val treesum : numtree -> int = <fun>
# let tt = Tree (Tree (Empty, 1, Empty), 3, Tree (Empty, 2,Empty));;
# treesum tt;;
- : int = 6
```

a function on lists

```
# let rec sum l = match l with [] -> 0 | e :: rest -> e + sum rest;;
val sum : int list -> int = <fun>
# sum [1;2;3;4];;
- : int = 10
```

Val treesum : numtree -> int;

```
let it = Tree (Tree (Empty, 1, Empty), 3, Tree (Empty, 2,Empty))

val treesum : numtree -> int
```

Lete rec treesum | match | with | 0 | e :: rest -> e + sum rest

function

pattern
puzzle: poly functional over lists

val map : ('a -> 'b) -> 'a list -> 'b list

solution

# let rec map f l = match l with [] -> [] | x :: xs -> (f x) :: (map f xs);

val map : ('a -> 'b) -> 'a list -> 'b list = <fun>

# map (fun x -> x * 2) [1; 2; 3]

- : int list = [2; 4; 6]
puzzle: user-defined poly datatypes

```
val treefold : 'a tree -> ('a * 'b * 'a -> 'a) -> 'a * 'b = <fun>

match t with
    | Empty -> b
    | Tree (left, v, right) -> f (treefold left, v, treefold right)
```

what is the type of treefold?

```
let rec treefold f b t =    match t with
    | Empty -> b
    | Tree (left, v, right) -> f (treefold left, v, treefold right);;
```

# type 'a tree

## a polymorphic tree

## a polymorphic tree

## a polymorphic tree
# let seed = ref 0;; val seed : int ref = {contents=0}

dereference

# !seed;;- : int = 0

assignment

# seed := 1;; - : unit = ()# !seed;;- : int = 1

side-effects

{val seed : int ref = contents=0} = Int

mutable cells

let seed = ref 0;;

#
write a function next which produces 0, 1, 2, etc.

takes no arguments

Puzzle
closures and cells

```
# let next = (let seed = ref 0 in function () -> seed := !seed + 1; !seed);;
val next : unit -> int = <fun>

# (next);;
val it : unit -> int = <fun>

# next ();;
val it : int = 1

# next ();;
val it : int = 2
```

```
lazy lists or 'streams'

Here is an example of a value that is not matched: Nil

Warning: this pattern-matching is not exhaustive.

Let il s = match s with Cons (x, f) -> f ();

Let il s = match s with Cons (x, f) -> f ();

Let il s = match s with Cons (x, f) -> f ();

Warning: this pattern-matching is not exhaustive.

Let il s = match s with Cons (x, f) -> f ();

Warning: this pattern-matching is not exhaustive.

Let il s = match s with Cons (x, f) -> f ();

Warning: this pattern-matching is not exhaustive.

Let il s = match s with Cons (x, f) -> f ();

Warning: this pattern-matching is not exhaustive.

Let il s = match s with Cons (x, f) -> f ();

Warning: this pattern-matching is not exhaustive.

Let il s = match s with Cons (x, f) -> f ();

Warning: this pattern-matching is not exhaustive.

Let il s = match s with Cons (x, f) -> f ();

Warning: this pattern-matching is not exhaustive.

Let il s = match s with Cons (x, f) -> f ();

Warning: this pattern-matching is not exhaustive.

Let il s = match s with Cons (x, f) -> f ();

Warning: this pattern-matching is not exhaustive.

Let il s = match s with Cons (x, f) -> f ();

Warning: this pattern-matching is not exhaustive.

Let il s = match s with Cons (x, f) -> f ();

Warning: this pattern-matching is not exhaustive.

Let il s = match s with Cons (x, f) -> f ();

Warning: this pattern-matching is not exhaustive.

Let il s = match s with Cons (x, f) -> f ();

Warning: this pattern-matching is not exhaustive.

Let il s = match s with Cons (x, f) -> f ();

Warning: this pattern-matching is not exhaustive.

Let il s = match s with Cons (x, f) -> f ();

Warning: this pattern-matching is not exhaustive.

Let il s = match s with Cons (x, f) -> f ();

Warning: this pattern-matching is not exhaustive.

Let il s = match s with Cons (x, f) -> f ();

Warning: this pattern-matching is not exhaustive.

Let il s = match s with Cons (x, f) -> f ();

Warning: this pattern-matching is not exhaustive.

Let il s = match s with Cons (x, f) -> f ();

Warning: this pattern-matching is not exhaustive.

Let il s = match s with Cons (x, f) -> f ();

Warning: this pattern-matching is not exhaustive.

Define a datatypes for streams
using streams

```ocaml
let rec from k = Cons (k, fun () -> from (k+1));;val from : int -> int stream = <fun>

# hd (tl (from 3));;val hd : int stream -> int = <fun>

- : int = 4

# (from 3) #

< : int stream = Cons (3, <fun>)

# (from 3) #

< : int stream = Cons (3, <fun>)

val from : int -> int stream = <fun>

let rec from k = Cons (k, fun () -> from (k+1));;val from : int -> int stream = <fun>
```
write a function that performs a depth-first traversal of a tree that gives result as a stream. For appending streams you can assume an infix function @ for appending streams.

Given

type 'a tree = Empty | Tree of 'a * 'a tree list

puzzle