CMSC330

Ocaml Features, Recursion, and Higher-order functions
Yesterday

• Ocaml intro
  – Lists
  – \texttt{let}
  – Types
  – Functions
  – Pattern matching
Today

• More Ocaml
  – Tuples
  – Recursion
  – Higher order functions
OCaml Functions Take One Argument

• Recall this example

```ocaml
let plus (x, y) = x + y;;
plus (3, 4);;
```

– It looks like you’re passing in two arguments

• Actually, you’re passing in a tuple instead

```ocaml
let plus t = match t with
  (x, y) = x + y;;
plus (3, 4);;
```

– And using pattern matching to extract its contents
Tuples

• **Constructed** using \((e_1, \ldots, e_n)\)
• **Deconstructed** using pattern matching
• Tuples are like C structs
  – But without field labels
  – Allocated on the heap
• Tuples can be heterogenous
  – Unlike lists, which must be homogenous
    – \((1, ["string1"; "string2"])) is a valid tuple
Tuples - examples

- let plusThree (x, y, z) = x+y+z
  let addOne (x, y, z) = (x+1, y+1, z+1)
  
  - plusThree (addOne (3,4,5)) = 15

- let sum ((a, b), c) = (a+c, b+c)
  
  - sum ((1, 2), 3) = (4,5)

- let plusFirstTwo (x::y::__:_, a) = (x+a, y+a)
  
  - plusFirstTwo ([1; 2; 3], 4) = (5,6)
Tuples – more examples

• let tls (::_:xs, :::ys) = (xs, ys)
  – tls ([1;2;3],[4;5;6;7]) = ([2;3],[5;6;7])

• Remember
  – Semicolon for lists
  – Comma for tuples

• Example
  – [1, 2] = [(1, 2)] = a list of size one
  – (1; 2) = a syntax error
Another tuple example

• Given
  - `let f l = match l with x::(_::y) -> (x,y)`

• What is the value of
  - `f [1;2;3;4]`

• Possibilities
  - `([1],[3])`
  - `(1,3)`
  - `(1,[3])`
  - `(1,4)`
  - `(1,[3;4])`
List and Tuple Types

• Tuple types use * to separate components

• Examples

- (1,2) : int * int
- (1,"string",3.5) : int * string * float
- (1, ["a";"b"], 'c') : int * string list * char
- [(1,2)] : (int * int) list
- [(1,2); (3, 4)] : (int * int) list
- [(1,2); (1,2,3)] : ERROR
Type Declarations

- **type** can be used to create new names for types
  - Useful for combinations of lists and tuples

**Examples**

- `type my_type = int * (int list)`
  
  `(3, [1; 2]) : my_type`

- `type my_type2 = int * char * (int * float)`
  
  `(3, ‘a’, (5, 3.0)) : my_type2`
Polymorphic Functions

• Some functions require specific list types
  - `let plusFirstTwo (x::y::_, a) = (x + a, y + a)`
  - `plusFirstTwo : int list * int -> (int * int)`
• But other functions work for a list of any type
  - `let hd (h::_) = h`
  - `hd [1; 2; 3]` (* returns 1 *)
  - `hd ["a"; "b"; "c"]` (* returns "a" *)
• These functions are polymorphic
Polymorphic Types

• OCaml gives such functions polymorphic types
  – \( \text{hd} : 'a \text{ list} \rightarrow 'a \)
  – Read as
    • Function takes a list of any element type \('a\)
    • And returns something of that type

• Example
  – \( \text{let tl (\_::t) = t} \)
    \( \text{tl} : 'a \text{ list} \rightarrow 'a \text{ list} \)
Polymorphic Types (cont.)

• More Examples
  
  - let swap (x, y) = (y, x)
    swap : 'a * 'b -> 'b * 'a

  - let tls (_::xs, _::ys) = (xs, ys)
    tls : 'a list * 'b list ->
    'a list * 'b list
Tuples Are a Fixed Size

- This OCaml definition
  ```ocaml
  # let foo x = match x with
  | (a, b) -> a + b
  | (a, b, c) -> a + b + c;;
  ```

- Would yield this error message
  - This pattern matches values of type 'a * 'b * 'c but is here used to match values of type 'd * 'e

- Tuples of different size have different types
  - Thus never more than one match case with tuples
Conditionals

• Use **if...then...else** just like C/Java
  – No parentheses and no end

```plaintext
if grade >= 90 then
  print_string "You got an A"
else if grade >= 80 then
  print_string "You got a B"
else if grade >= 70 then
  print_string "You got a C"
else
  print_string "You’re not doing so well"
```

Conditionals (cont.)

- In OCaml, conditionals return a result
  - The value of whichever branch is true/false

```ocaml
# if 7 > 42 then "hello" else goodbye;;
- : string = "goodbye"

# let x = if true then 3 else 4;;
x : int = 3

# if false then 3 else 3.0;;
This expression has type float but is here used with type int
```
The Factorial Function

• Using conditionals & functions
  – Can you write fact, the factorial function?

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

• Notice no return statements
  – This is pretty much how it needs to be written
let rec

• The rec part means “define a recursive function”
• Let vs. let rec
  - let x = e1 in e2
    - x in scope within e2
  - let rec x = e1 in e2
    - x in scope within e2 and e1
• Why use let rec?
  - If you used let instead of let rec to define fact

```plaintext
let fact n =
  if n = 0 then 1
  else n * fact (n-1) in el
```

Fact is not bound here!
Examples - semicolon

• Definition
  - e1 ; e2 (* evaluate e1, evaluate e2, return e2)

• 1 ; 2 ;;
  - (* 2 - value of 2\textsuperscript{nd} expression is returned *)

• (1 + 2) ; 4 ;;
  - (* 4 - value of 2\textsuperscript{nd} expression is returned *)

• 1 + (2 ; 4) ;;
  - (* 5 - value of 2\textsuperscript{nd} expression is returned to 1 + *)

• 1 + 2 ; 4 ;;
  - (* 4 - since + has higher precedence than ; *)
Examples - `let`

- `x;;`
  - (* Unbound value x *)

- `let x = 1 in x + 1;;`
  - (* 2 *)

- `let x = x in x + 1;;`
  - (* Unbound value x *)
Examples – *let* (cont.)

- `let x = 1 in (x + 1 ; x) ;;`
  - (* 1 – ; has higher precedence than let ... in *)

- `(let x = 1 in x + 1) ; x;;`
  - (* Unbound value x *)

- `let x = 4 in (let x = x + 1 in x);`
  - (* 5 *)
let – more examples

- let f n = 10;;
  let f n =
      if n = 0 then 1
      else n * f (n - 1);;

  - f 0;; (* = 1 *)
  - f 1;; (* = 10 *)

- let f x = f x;;
  - (* Unbound value f *)
Recursion

• Recursion is essentially the only way to iterate
  – The only way we’re going to talk about, anyway
  – Feature of functional programming languages

• Another example

```plaintext
let rec print_up_to (n, m) =
  print_int n; print_string "\n";
  if n < m then print_up_to (n + 1, m)
```
Lists and recursion

• Lists have a recursive structure
  – And so most functions over lists will be recursive

    let rec length l = match l with
    []   -> 0
    | (_::t) -> 1 + (length t)

  – This is just like an inductive definition
    • The length of the empty list is zero
    • The length of a nonempty list is 1 plus the length of the tail

  – Type of length?
Examples – recursive functions

• sum li (* sum of elements in li *)
  
  \[
  \text{let rec sum \textit{li} = match \textit{li} with} \\
  \hspace{90pt} [] \rightarrow 0 \\
  \hspace{90pt} (x::xs) \rightarrow x + (\text{sum xs})
  \]

• negate li (* negate elements in list *)

  \[
  \text{let rec negate \textit{li} = match \textit{li} with} \\
  \hspace{90pt} [] \rightarrow [] \\
  \hspace{90pt} (x::xs) \rightarrow (-x) :: (\text{negate xs})
  \]
Example – recursive functions

• last li (* last element of li *)
  
  `let rec last li = match li with`
  
  `  [x] -> x`
  
  `| (_::xs) -> last xs`

• append (li, m)
  (* list containing all elements in list li followed by all
elements in list m *)

  `let rec append (li, m) = match li with`
  
  `  [] -> m`
  
  `| (x::xs) -> x::(append (xs, m))`
Example – recursive functions

• rev li (* reverse list; hint: use append *)

```ocaml
let rec rev li = match li with
  | [] -> []
  | (x::xs) -> append ((rev xs), [x])
```

- `rev` takes $O(n^2)$ time. Can you do better?
A clever version of reverse

```ml
let rec rev_helper (li, a) = match li with
  | [] -> a
  | (x::xs) -> rev_helper (xs, (x::a))
let rev li = rev_helper (li, [])
```

• Let’s give it a try

```
rev [1; 2; 3] →
rev_helper ([1;2;3], []) →
rev_helper ([2;3], [1]) →
rev_helper ([3], [2;1]) →
rev_helper ([], [3;2;1]) →
[3;2;1]
```
Example – recursive functions

- flattenPairs li (* ('a * 'a) list -> 'a list *)
  
  \[
  \text{let rec flattenPairs li} = \text{match li with}
  \]
  
  \[
  [] \rightarrow []
  \]
  
  \[
  | ((a, b)::t)
  \]
  
  \[
  \rightarrow a :: b :: (\text{flattenPairs t})
  \]

- take (n, li) (* return first n elements of li *)
  
  \[
  \text{let rec take (n, li) =}
  \]
  
  \[
  \text{if n} = 0 \text{ then []}
  \]
  
  \[
  \text{else match li with}
  \]
  
  \[
  [] \rightarrow []
  \]
  
  \[
  | (x::xs) \rightarrow x :: (\text{take (n-1, xs)})
  \]
Working with lists

• Several of these examples have the same flavor
  – Walk through the list and do something to every element
  – Walk through the list and keep track of something
• Recall the following example code from Ruby:

  a = [1,2,3,4,5]
  b = a.collect { |x| -x }

  – Here we passed a code block into the `collect` method
  – Wouldn’t it be nice to do the same in OCaml?
Higher order functions

• In OCaml you can pass functions as arguments, and return functions as results

```ocaml
let plus_three x = x + 3
let twice (f, z) = f (f z)
    (* twice : ('a->'a) * 'a -> 'a *)
twice (plus_three, 5)

let plus_four x = x + 4
let pick_fn n =
    if n > 0 then plus_three else plus_four
(pick_fn 5) 0
(* pick_fn : int -> (int->int) *)
```
The **map** function

- Let’s write the **map** function (just like Ruby’s **collect**)
  - Takes a function and a list, applies the function to each element of the list, and returns a list of the results

```ocaml
let rec map (f, l) = match l with
    [] -> []
  | (h::t) -> (f h)::(map (f, t))
```

```ocaml
let add_one x = x + 1
let negate x = -x
map (add_one, [1; 2; 3])
map (negate, [9; -5; 0])
```
The \textbf{map} function (cont.)

- What is the type of the \textbf{map} function?

\begin{verbatim}
let rec map (f, l) = match l with
    []    -> []
| (h::t) -> (f h)::(map (f, t))
\end{verbatim}

\[
('a -> 'b) * 'a list -> 'b list
\]

\textbf{map} function
Anonymous Functions

• Passing functions around is very common
  – So often we don’t want to bother to give them names

• Use `fun` to make a function with no name

```
fun x -> x + 3
```

\[
twice ((fun x -> x + 3), 5) = 11
\]
\[
map ((fun x -> x + 1), [1; 2; 3]) = [2; 3; 4]
\]
Pattern matching with \texttt{fun}

- \texttt{match} can be used within \texttt{fun}

\begin{verbatim}
map ((fun l -> match l with (h::_) -> h),
    [ [1; 2; 3]; [4; 5; 6; 7]; [8; 9] ])
\end{verbatim}

- But use named functions for complicated matches
- May use standard pattern matching abbreviations

\begin{verbatim}
map ((fun (x, y) -> x+y), [(1,2); (3,4)])
\end{verbatim}

= \[3;7\]
All functions are anonymous

- Functions are first-class, so you can bind them to other names as you like
  
  ```
  let f x = x + 3
  let g = f
  g 5
  ```

- In fact, `let` for functions is just shorthand
  
  ```
  let f x = body
  ```
  stands for
  
  ```
  let f = fun x -> body
  ```
Examples – anonymous functions

• let next \( x = x + 1 \)
  – Short for let next = fun \( x \rightarrow x + 1 \)

• let plus \((x, y) = x + y \)
  – Short for let plus = fun \((x, y) \rightarrow x + y\)
  – Which is short for
    
    let plus = fun \( z \rightarrow (\text{match } z \text{ with } (x, y) \rightarrow x + y)\)
Examples – anonymous functions

- let rec fact n =
  if n = 0 then 1
  else n * fact (n-1)

  – Short for let rec fact = fun n ->
    (if n = 0 then 1
     else n * fact (n-1))
The **fold** function

- **Common pattern**
  - Iterate through list and apply function to each element, keeping track of partial results computed so far

```ml
let rec fold (f, a, l) = match l with
  [] -> a
| (h::t) -> fold (f, f (a, h), t)
```

- `a` = “accumulator”
- Usually called **fold left** to remind us that `f` takes the accumulator as its first argument

- **What's the type of **fold**?**
fold example

let rec fold (f, a, l) = match l with
    [] -> a
  | (h::t) -> fold (f, f (a, h), t)

let add (a, x) = a + x
fold (add, 0, [1; 2; 3; 4]) →
fold (add, 1, [2; 3; 4]) →
fold (add, 3, [3; 4]) →
fold (add, 6, [4]) →
fold (add, 10, []) →
10

We just built the sum function!
Another **fold** example

```ocaml
let rec fold (f, a, l) = match l with
  | []  -> a
  | h::t -> fold (f, f (a, h), t)
```

```ocaml
let next (a, _) = a + 1
fold (next, 0, [2; 3; 4; 5]) →
fold (next, 1, [3; 4; 5]) →
fold (next, 2, [4; 5]) →
fold (next, 3, [5]) →
fold (next, 4, []) →
4
```

We just built the **length** function!
Using **fold** to build reverse

```ocaml
let rec fold (f, a, l) = match l with
    [] -> a
  | (h::t) -> fold (f, f (a, h), t)

let prepend (a, x) = x::a
fold (prepend, [], [1; 2; 3; 4]) →
fold (prepend, [1], [2; 3; 4]) →
fold (prepend, [2; 1], [3; 4]) →
fold (prepend, [3; 2; 1], [4]) →
fold (prepend, [4; 3; 2; 1], []) →
[4; 3; 2; 1]
```

• Can you build the **rev** function with **fold**?
The call stack in Java/C++/etc

```c
void f(void) {
    int x;
    x = g(3);
}

int g(int x) {
    int y;
    y = h(x);
    return y;
}

int h (int z) {
    return z + 1;
}

int main(){
    f();
    return 0;
}
```

```
x 4
x 3
y 4
z 3
f
```
Nested functions

- In OCaml, you can define functions anywhere
  - Even inside of other functions

```ocaml
let sum l = fold ((fun (a, x) -> a + x), 0, l)

let pick_one n = if n > 0 then (fun x -> x + 1)
else (fun x -> x - 1)
(pick_one -5) 6 (* returns 5 *)
```
Nested functions (cont.)

• You can also use `let` to define functions inside of other functions

```plaintext
let sum l = 
    let add (a, x) = a + x in
    fold (add, 0, l)

let pick_one n = 
    let add_one x = x + 1 in
    let sub_one x = x - 1 in
    if n > 0 then add_one else sub_one
```
How about this?

```ocaml
let addN (n, l) =
  let add x = n + x in
  map (add, l)
```

- (Equivalent to...)

```ocaml
let addN (n, l) =
  map ((fun x -> n + x), l)
```

Accessing variable from outer scope
Consider the call stack again

```
let map (f, n) = match n with
    [] -> []
| (h::t) -> (f h)::(map (f, t))

let addN (n, l) =
    let add x = n + x in
    map (add, l)
```

\[ \text{addN (3, [1; 2; 3])} \]

- Uh oh...how does `add` know the value of `n`?
  - **Dynamic scoping**: it reads it off the stack
    - The language could do this, but can be confusing (see above)
  - OCaml uses **static scoping** like C, C++, Java, and Ruby
Static scoping

• In static or lexical scoping, (nonlocal) names refer to their nearest binding in the program text
  – Going from inner to outer scope
  – In our example, `add` refers to `addN`’s `n`
  – C example:

```c
int x;
void f() { x = 3; }
void g() { char *x = "hello"; f(); }
```

Refers to the `x` at file scope – that’s the nearest `x` going from inner scope to outer scope in the source code
Summary

• Tuples
• Recursion
• Higher order functions
Discussion tomorrow

• Return Quizzes
• Ocaml examples (posted on schedule)