OCaml 2
Higher Order Functions

Examples With Tuples

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

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

3. let plusFirstTwo (x::y::_, a) = (x + a, y + a)
   * plusFirstTwo ([1; 2; 3], 4) = (5, 6)

4. 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
* [1; 2] = [(1, 2)] = a list of size one
* (1; 2) = Warning: This expression should have type unit

Another Example

1. let f l = match l with x:(_::y) -> (x,y)
   What is f [1;2;3;4]
   Possibilities:
   (1,[3])
   (1,3)
   (1,[3])
   (1,4)
   (1,[3;4])

Tuples

- Constructed using (e1, ..., en)
- Deconstructed using pattern matching
  - Patterns involve parens and commas, e.g., (p1,p2,...)
- 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
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`
  - Because the first list element has type int * int, but the second has type int * int - list elements must all be of the same type

Polymorphic Types

- Some functions we saw require specific list types
  - `let plusFirstTwo (x::y::_, a) = (x + a, y + a)`
  - `plusFirstTwo : int list * int -> (int * int)
- But other functions work for any list
  - `let hd (h::_) = h`
  - `hd [1; 2; 3]` (* returns 1 *)
- OCaml gives such functions polymorphic types
  - `hd : 'a list -> 'a`
    - this says the function takes a list of any element type 'a, and returns something of that type

Examples Of Polymorphic Types

- `let tl (_,::t) = t`
  - `tl : 'a list -> 'a list`
- `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`
- `let eq (x,y) = x = y`
  - `eq : 'a * 'a -> bool`
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
  ```ocaml
  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 like C/Java/Ruby
  ```ocaml
  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
  - Like ?: in C, C++, and Java
    ```ocaml
    # if 7 > 42 then "hello" else "goodbye";;
    : string = "goodbye"
    # let x = if true then 3 else 4;;
    : 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?
    ```ocaml
    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 \( x = e_1 \) in \( e_2 \) \( x \) in scope within \( e_2 \)
  - let rec \( x = e_1 \) in \( e_2 \) \( x \) in scope within \( e_2 \) and \( e_1 \)
- Why use let rec?
  - If you used let instead of let rec to define fact
    ```
    let fact n = 
      if n = 0 then 1
      else n * fact (n - 1)
    in e2
    ```
  - Fact is not bound here!

Recursion = Looping

- Recursion is essentially the only way to iterate
  - (The only way we’re going to talk about)
- Another example
  ```
  let rec print_up_to (n, m) = 
    print_int n; print_string "\n";
    if n < m then print_up_to (n + 1, m)
  ```

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 \( f x = \) ... \( f \) ... in ... \( f \) ... 
    - (* Unbound value \( f \) *)
  - let rec \( f x = \) ... \( f \) ... in ... \( f \) ... 
    - (* Bound value \( f \) *)

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 \( \text{length} \)?
More Examples

• sum l (* sum of elts in l *)
  let rec sum l = match l with
  | [] -> 0
  | (x::xs) -> x + (sum xs)

• negate l (* negate elements in list *)
  let rec negate l = match l with
  | [] -> []
  | (x::xs) -> (-x) :: (negate xs)

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

More Examples (cont.)

(* return a list containing all the elements in the
list l followed by all the elements in list m *)

• append (l, m)
  let rec append (l, m) = match l with
  | [] -> m
  | (x::xs) -> x :: (append (xs, m))

• rev l (* reverse list; hint: use append *)
  let rec rev l = match l with
  | [] -> []
  | (x::xs) -> append ((rev xs), [x])

• rev takes O(n^2) time. Can you do better?

A Clever Version of Reverse

let rec rev_helper (l, a) = match l with
  | [] -> a
  | (x::xs) -> rev_helper (xs, (x::a))

let rev l = rev_helper (l, [])

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]

More Examples

• flattenPairs l (* ('a * 'a) list -> 'a list *)
  let rec flattenPairs l = match l with
  | [] -> []
  | ((a, b)::t) -> a :: b :: (flattenPairs t)

• take (n, l) (* return first n elts of l *)
  let rec take (n, l) = if n = 0 then []
                          else match l with
                              | [] -> []
                              | (x::xs) -> x :: (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:

```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?

Anonymous Functions

Recall code blocks in Ruby

```ruby
(1..10).each { |x| print x }
```

- Here, we can think of `{ |x| print x }` as a function

We can do this (and more) in OCaml

```ocaml
let rec range_each (i,j) f =
  if i > j then ()
  else
    let _ = f i in (* ignore result *)
    range_each (i+1,j) f

range_each (1,10) (fun x -> print_int x)
```

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

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

Parameter Body

```ocaml
(fun x -> x + 3) 5 = 8
```

All Functions Are Anonymous

Functions are first-class, so you can bind them to other names as you like

```ocaml
let f x = x + 3
let g = f

let f x = body
```

- In fact, `let` for functions is just shorthand

```ocaml
let f = fun x -> body
```
**Examples**

- let next x = x + 1
  - Short for let next = fun x -> x + 1

- let plus (x, y) = x + y
  - Short for let plus = fun (x, y) -> x + y
  - Which is short for
    - let plus = fun z -> (match z with (x, y) -> x + y)

- 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))

**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 plus_three 5
  twice ('a->'a) -> 'a -> 'a
  ```

  ```ocaml
  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)
  ```

**Currying**

- We just saw a way for a function to take multiple arguments
  - The function consumes one argument at a time, returning a function that takes the rest

- This is called currying the function
  - Named after the logician Haskell B. Curry
  - But Schönfinkel and Frege discovered it
    - So it should probably be called Schönfinkelizing or Fregging

**Curried Functions In OCaml**

- OCaml has a really simple syntax for currying

  ```ocaml
  let add x y = x + y
  ```

  - This is identical to all of the following:

    ```ocaml
    let add = (fun x -> (fun y -> x + y))
    let add = (fun x y -> x + y)
    let add x = (fun y -> x+y)
    ```

  - Thus:
    - `add` has type `int -> (int -> int)`
    - `add 3` has type `int -> int`
      - `add 3` is a function that adds 3 to its argument
    - `(add 3) 4 = 7`

- This works for any number of arguments
Curried Functions In OCaml (cont.)

- Because currying is so common, OCaml uses the following conventions:
  - `->` associates to the right
    - Thus `int -> int -> int` is the same as `int -> (int -> int)`
  - function application associates to the left
    - Thus `add 3 4` is the same as `(add 3) 4`

Mental Shorthand

- You can think of curried types as defining multi-argument functions
  - Type `int -> float -> float` is a function that takes an `int` and a `float` and returns a `float`
  - Type `int -> int -> int -> int` is a function that takes three `ints` and returns an `int`

- The bonus is that you can partially apply the function to some of its arguments
  - And apply that to the rest of the arguments later

Another Example Of Currying

- A curried add function with three arguments:

  ```ocaml
  let add_th x y z = x + y + z
  let add_th x = (fun y -> (fun z -> x+y+z))
  ```

- The same as

  ```ocaml
  let add_th x = (fun y -> (fun z -> x+y+z))
  ```

- Then...

  - `add_th` has type `int -> (int -> (int -> int))`
  - `add_th 4` has type `int -> (int -> int)`
  - `add_th 4 5` has type `int -> int`
  - `add_th 4 5 6` is 15

Implementing this is Challenging!

- Implementing functions that return other functions requires a clever data structure called a closure
  - We’ll see how these are implemented later

- In the meantime, we will explore using higher order functions, and then discuss how they are implemented
The Map Function

- Let's write the map function (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

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

- Type of map?

---

Pattern Matching With Fun

- match can be used within fun

```ml
map (fun l -> match l with (h::_) -> h)
[ [1; 2; 3]; [4; 5; 6; 7]; [8; 9] ]
= [1; 4; 8]
```

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

```
map (fun (x, y) -> x+y) [(1,2); (3,4)]
= [3; 7]
```

---

The Fold Function

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

```
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?
  - = (a -> b -> a) -> a
Example

```ocaml
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 Example

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

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

Can you build the `reverse` function with fold?

```

```ocaml
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]
```