This Lecture

- Higher order functions
  - Map, fold
- Static & dynamic scoping
- Environments & closures
- Currying

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) = 11
// twice : (a->a) * a -> a

let plus_four x = x + 4
let pick_fn n = if n > 0 then plus_three else plus_four
(pick_fn 5) 0 = 3
// 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))

let add_one x = x + 1
let negate x = -x
map (add_one, [1; 2; 3]) = [2; 3; 4]
map (negate, [9; -5; 0]) = [-9; 5; 0]
```

The map Function (cont.)

- What is the type of the map function?

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

('a -> 'b) * 'a list -> 'b list
```

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
```

```ocaml
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}

\[
\text{map } ((\texttt{fun } l \rightarrow \texttt{match } l \texttt{ with })(h::__:h)) = \text{[1; 4; 8]}
\]

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

\[\text{map } ((\texttt{fun } (x, y) \rightarrow x + y), \text{[(1,2); (3,4)]}) = \text{[3; 7]}
\]

All Functions Are Anonymous

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

\[
\text{let } f x = x + 3
\]

\[
\text{let } g = f
\]

\[
g 5 = 8
\]

- In fact, \texttt{let} for functions is just shorthand

\[\downarrow \text{ stands for}
\]

\[\text{let } f = \texttt{fun } x \rightarrow \text{body}
\]

Examples – Anonymous Functions

- \texttt{let next} \( x = x + 1 \)
  - Short for \texttt{let next = fun} \( x \rightarrow x + 1 \)

- \texttt{let plus} \( (x, y) = x + y \)
  - Short for \texttt{let plus = fun} \( (x, y) \rightarrow x + y \)
  - Which is short for

\[
\text{let plus } = \texttt{fun } z \rightarrow
\]

\[
\text{match } z \texttt{ with } (x, y) \rightarrow x + y
\]

The fold Function

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

\[
\text{let rec fold } (f, a, l) = \texttt{match } l \texttt{ with }\]

\[
\begin{align*}
| & \rightarrow a \\
| h::t & \rightarrow \text{fold } (f, f (a, h), t)
\end{align*}
\]

- \( a = \text{“accumulator”} \)
- Usually called \texttt{fold} \texttt{left} to remind us that \( f \) takes the accumulator as its first argument

- What's the type of \texttt{fold}?

\[
= (\texttt{a} \times \texttt{b} \rightarrow \texttt{a}) \times (\texttt{a} \times \texttt{b} \texttt{ list} \rightarrow \texttt{a})
\]

Example

\[
\text{let rec fold } (f, a, l) = \texttt{match } l \texttt{ with }\]

\[
\begin{align*}
| & \rightarrow a \\
| h::t & \rightarrow \text{fold } (f, f (a, h), t)
\end{align*}
\]

\[
\text{let add } (a, x) = a + x
\]

\[
\begin{align*}
\text{fold } (\texttt{add}, 0, [1; 2; 3; 4]) & \rightarrow \\
\text{fold } (\texttt{add}, 1, [2; 3; 4]) & \rightarrow \\
\text{fold } (\texttt{add}, 3, [3; 4]) & \rightarrow \\
\text{fold } (\texttt{add}, 6, [4]) & \rightarrow \\
\text{fold } (\texttt{add}, 10, [1]) & \rightarrow 10
\end{align*}
\]

We just built the \texttt{sum} function!
**Another Example**

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

Let next \( a, _ \) = \( a + 1 \)

Fold \( \text{next}, 0, [2; 3; 4; 5] \) ->
- Fold \( \text{next}, 1, [3; 4; 5] \) ->
- Fold \( \text{next}, 2, [4; 5] \) ->
- Fold \( \text{next}, 3, [5] \) ->
- Fold \( \text{next}, 4, [] \) ->

We just built the length function!

**Using fold to Build rev**

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?
  - Let prepend \( (a, x) = x::a \)
  - Fold \( \text{prepend}, [1, [1; 2; 3; 4]] \) ->
  - Fold \( \text{prepend}, [1, [2; 1; 3; 4]] \) ->
  - Fold \( \text{prepend}, [3; 2; 1], [4] \) ->
  - Fold \( \text{prepend}, [4; 3; 2; 1], [] \) ->

**The Call Stack in C/Java/etc.**

```
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;
}
```

**Nested Functions**

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

  ```
  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

  ```
  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?**

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

- (Equivalent to...)

```
let addN (n, l) =
map ((fun x -> n + x), l)
```
Consider the Call Stack Again

```ocaml
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)
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

Returned Functions

- As we saw, in OCaml a function can return another function as a result
  - So consider the following example

```ocaml
let addN n = (fun x -> x + n)
addN 3 4 (* returns 7 *)
```

- When the anonymous function is called, `n` isn’t even on the stack any more!
  - We need some way to keep `n` around after `addN` returns

Environments and Closures

- An environment is a mapping from variable names to values
  - Just like a stack frame

- A closure is a pair `(f, e)` consisting of function code `f` and an environment `e`

- When you invoke a closure, `f` is evaluated using `e` to look up variable bindings

Example – Closure 1

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

```
(add 3) 4  -> 4  -> 3 + 4  -> 7
```

Function  →  Environment

Example – Closure 2

```ocaml
let mult_sum (x, y) = let z = x + y in
fun w -> w * z
```

```
(mult_sum (3, 4)) 5  -> 5  -> 5 * 7  -> 35
```

실행 시점에 따라 각 변수의 값이 정해진다. 이는 영역 기반의 상호작용을 가능하게 한다.
Example – Closure 3

```ml
let twice (n, y) = let f x = x + n in f (f y)
twice (3, 4)
```

twice (3, 4) \(\mapsto\) \(<\text{cl}>\) \((<\text{cl}>\) 4) \(\mapsto\) \(<\text{cl}>\) 7 \(\mapsto\) 10

Example – Closure 4

```ml
let add x = (fun y -> (fun z -> x + y + z)) (((add 1) 2) 3)
```

\(((\text{add}\ 1)\ 2)\ 3\) \(\mapsto\) \((<\text{cl}>\) 2) \(\mapsto\) \((<\text{cl}>\) 3) \(\mapsto\) 1+2+3

Currying

- We just saw another way for a function to take multiple arguments
  - The function consumes one argument at a time, creating closures until all the arguments are available
- 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
  ```ml
  let add x y = x + y
  ```
  - This is identical to all of the following
    ```ml
    let add = (fun x -> (fun y -> x + y))
    let add = (fun x y -> x + y)
    let add x = (fun y -> x + y)
    ```

Curried Functions in OCaml (cont.)

- What is the type of `add`?
  ```ml
  let add x y = x + y
  ```

- Answer
  - `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.)

- Currying is so common, OCaml uses the following conventions
  ```ml
  let add x y = x + y
  ```

- `->` associates to the right
  - `int -> int -> int` is the same as `int -> (int -> int)`
- Function application `()` associates to the left
  - `(add 3) 4` is the same as `(add 3) 4`
Another Example of Currying

- A curried add function with three arguments
  
  \[
  \text{let add_th x y z = x + y + z}
  \]

  is the same as
  
  \[
  \text{let add_th x = (fun y -> (fun z -> x+y+z))}
  \]

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

Currying and the `map` Function

- New Map
  
  \[
  \text{let rec map f l = match l with}
  \]

  \[
  \begin{align*}
  & | [] -> [] \\
  & | (h::t) -> (f h)::(map f t)
  \end{align*}
  \]

- Examples
  
  \[
  \text{let negate x = -x}
  \]

  \[
  \text{map negate [1; 2; 3] (* [-1; -2; -3] *)}
  \]

  \[
  \text{let negate_list = map negate}
  \]

  \[
  \text{negate_list [-1; -2; -3] (* [1; 2; 3] *)}
  \]

  \[
  \text{let sum_pair_l = map (fun (a, b) -> a + b)}
  \]

  \[
  \text{sum_pair_l[(1, 2); (3, 4)] (* [3; 7] *)}
  \]

- What is the type of this form of `map`?
  
  \[
  \text{('a -> 'b) -> 'a list -> 'b list}
  \]

Currying and the `fold` Function

- New Fold
  
  \[
  \text{let rec fold f a l = match l with}
  \]

  \[
  \begin{align*}
  & | [] -> a \\
  & | (h::t) -> fold f (f a h) t
  \end{align*}
  \]

- Examples
  
  \[
  \text{let add x y = x + y}
  \]

  \[
  \text{fold 0 [1; 2; 3] (* 6 *)}
  \]

  \[
  \text{let sum = fold add 0}
  \]

  \[
  \text{sum [1; 2; 3] (* 6 *)}
  \]

  \[
  \text{let next n_ = n + 1}
  \]

  \[
  \text{let len = fold next 0 (* len not polymorphic! *)}
  \]

  \[
  \text{len [4; 5; 6; 7; 8] (* 5 *)}
  \]

- What is the type of this form of `fold`?
  
  \[
  \text{('a -> 'b -> 'a) -> 'a -> 'b list -> 'a}
  \]

Another Convention

- Since functions are curried, `function` can often be used instead of `match`
  
  - `function` declares anonymous function w/ one argument
  
  - Instead of
    
    \[
    \text{let rec sum_l = match l with}
    \]

    \[
    \begin{align*}
    & | [] -> 0 \\
    & | (h::t) -> h + (sum t)
    \end{align*}
    \]

  - It could be written
    
    \[
    \text{let rec sum_l = function}
    \]

    \[
    \begin{align*}
    & | [] -> 0 \\
    & | (h::t) -> h + (sum t)
    \end{align*}
    \]

Another Convention (cont.)

- Instead of
  
  \[
  \text{let rec map f l = match l with}
  \]

  \[
  \begin{align*}
  & | [] -> [] \\
  & | (h::t) -> (f h)::(map f t)
  \end{align*}
  \]

- It could be written
  
  \[
  \text{let rec map f = function}
  \]

  \[
  \begin{align*}
  & | [] -> [] \\
  & | (h::t) -> (f h)::(map f t)
  \end{align*}
  \]
Currying is Standard in OCaml

- Pretty much all functions are curried
  - Like the standard library `map`, `fold`, etc.
  - See `/usr/local/ocaml/lib/ocaml` on `linuxlab`
    - In particular, look at the file `list.ml` for standard list functions
    - Access these functions using `List.<fn name>`
    - E.g., `List.hd List.length List.map`

- OCaml plays a lot of tricks to avoid creating closures and to avoid allocating on the heap
  - It's unnecessary much of the time, since functions are usually called with all arguments