Recursion = Looping

- 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

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

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

Examples – Recursive Functions

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

- append (l, m)
  ```ocaml
  (* list containing all elements in list l followed by all elements in list m *)
  let rec append (l, m) = match l with
      [] -> m
    | (x::xs) -> x::(append (xs, m))
  ```

Examples – Recursive Functions

- rev l (* reverse list; hint: use append *)
  ```ocaml
  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

```ocaml
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

```ocaml
rev [1; 2; 3]
```

Examples – Recursive Functions

```ocaml
> 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 elements 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
# Here we passed a code block into the collect method
# Wouldn't it be nice to do the same in OCaml?
```

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 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]
```

Higher-Order Functions

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

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))
```

```ocaml
('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

\[
\text{fun } x \rightarrow x + 3
\]

\[
\begin{align*}
twice \ ((\text{fun } x \rightarrow x + 3), 5) &= 11 \\
\text{map } \{(\text{fun } x \rightarrow x+1), [1; 2; 3]\} &= [2; 3; 4]
\end{align*}
\]

Pattern Matching with fun

- `match` can be used within `fun`

\[
\begin{align*}
\text{map } \{(\text{fun } l \rightarrow \text{match } l \text{ with } (h::\_ \rightarrow h), \ [ [1; 2; 3]; [4; 5; 6; 7]; [8; 9] ]\} &= [1; 4; 8]
\end{align*}
\]

- But use named functions for complicated matches

- May use standard pattern matching abbreviations

\[
\begin{align*}
\text{map } \{(\text{fun } (x, y) \rightarrow x+y), [(1,2); (3,4)]\} &= [3; 7]
\end{align*}
\]

All Functions Are Anonymous

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

\[
\begin{align*}
\text{let } f x &= x + 3 \\
\text{let } g &= f \\
g 5 &= 8
\end{align*}
\]

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

\[
\begin{align*}
\text{let } f x &= \text{body} \\
\phantom{\text{let } f x} &= \text{fun } x \rightarrow \text{body}
\end{align*}
\]

Examples – Anonymous Functions

- `let rec` `fact n =`
  - `if n = 0 then 1 else n * fact (n-1)`
  - Short for `let rec fact = fun n ->` \[ \text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact } (n-1) \]

Examples – Anonymous Functions

- `let rec` `fact n =`
  - `if n = 0 then 1 else n * fact (n-1)`
  - Short for `let rec fact = fun n ->` \[ \text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact } (n-1) \]

The fold Function

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

\[
\begin{align*}
\text{let rec } \text{fold } (f, a, l) &= \text{match } l \text{ with } \\
& \{ [] \rightarrow a \\
& | (h::t) \rightarrow f (a, h), t \}
\end{align*}
\]

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

- What’s the type of `fold`?

\[
\begin{align*}
\text{let rec } \text{fold } (f, a, l) &= \text{match } l \text{ with } \\
& \{ [] \rightarrow a \\
& | (h::t) \rightarrow f (a, h), t \}
\end{align*}
\]

\[
\begin{align*}
\text{let rec } \text{fold } (f, a, l) &= \text{match } l \text{ with } \\
& \{ [] \rightarrow a \\
& | (h::t) \rightarrow f (a, h), t \}
\end{align*}
\]

\[
\begin{align*}
\text{let rec } \text{fold } (f, a, l) &= \text{match } l \text{ with } \\
& \{ [] \rightarrow a \\
& | (h::t) \rightarrow f (a, h), t \}
\end{align*}
\]

\[
\begin{align*}
\text{let rec } \text{fold } (f, a, l) &= \text{match } l \text{ with } \\
& \{ [] \rightarrow a \\
& | (h::t) \rightarrow f (a, h), t \}
\end{align*}
\]

\[
\begin{align*}
\text{let rec } \text{fold } (f, a, l) &= \text{match } l \text{ with } \\
& \{ [] \rightarrow a \\
& | (h::t) \rightarrow f (a, h), t \}
\end{align*}
\]

\[
\begin{align*}
\text{let rec } \text{fold } (f, a, l) &= \text{match } l \text{ with } \\
& \{ [] \rightarrow a \\
& | (h::t) \rightarrow f (a, h), t \}
\end{align*}
\]

\[
\begin{align*}
\text{let rec } \text{fold } (f, a, l) &= \text{match } l \text{ with } \\
& \{ [] \rightarrow a \\
& | (h::t) \rightarrow f (a, h), t \}
\end{align*}
\]

\[
\begin{align*}
\text{let rec } \text{fold } (f, a, l) &= \text{match } l \text{ with } \\
& \{ [] \rightarrow a \\
& | (h::t) \rightarrow f (a, h), t \}
\end{align*}
\]
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, j] = 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 rev

```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 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]
```

The Call Stack in C/Java/etc.

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

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

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

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

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)
```

- You can also use let to define functions inside of other functions

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

```ocaml
let next (a, j) = 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
```

```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]
```

```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]
```

```ocaml
let pick_one n =
  if n > 0 then add_one else sub_one
```

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

let next (a, j) = 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
```
How About This?

\[
\text{let addN (n, l) =}
\begin{align*}
\text{let add } x &= n + x \text{ in } \\
\text{map } (add, l)
\end{align*}
\]

(Equivalent to...)

\[
\text{let addN (n, l) =}
\begin{align*}
\text{map } ((\text{fun } x \rightarrow n + x), l)
\end{align*}
\]

Accessing variable from outer scope

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

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)
```

```
addN (3, [1; 2; 3])
```

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:

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

- 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