# CMSC 330: Organization of Programming Languages

Functional Programming with OCaml, con't.



## 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

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

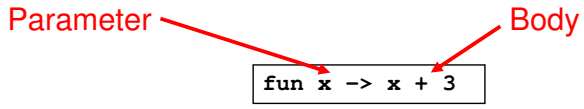
```
let add_one x = x + 1
let negate x = -x
map (add_one, [1; 2; 3])
map (negate, [9; -5; 0])
```

- Type of **map**?



## Anonymous Functions

- Passing functions around is very common, so we often don't want to bother to give them names
- Use fun to make a function with no name




```
map ((fun x -> x + 13), [1; 2; 3])
twice ((fun x -> x + 2), 4)
```



## Pattern Matching with fun

- match can be used within fun

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

  - for complicated matches, though, use named functions

- Standard pattern matching abbreviation can be used

```
map ((fun (x, y) -> x + y), [(1, 2); (3, 4)])
                            (* [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  (* returns 8 *)`

- let for functions is just a shorthand
  - `let f x = body` stands for
  - `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))`



## More Higher-Order Functions- the fold Function

- A common pattern is to iterate through a list and apply a function to each element, keeping track at the same time of the 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"
  - this is usually called "fold left" to remind us that f takes the accumulator as its first argument
- What's the type of `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 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 (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

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

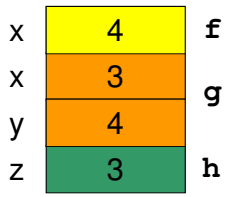
- Can you build the reverse function with fold?

```
let prepend (a, x) = x::a
fold (prepend, [], [5; 6; 7; 8]) →
fold (prepend, [5], [6; 7; 8]) →
fold (prepend, [6; 5], [7; 8]) →
fold (prepend, [7; 6; 5], [8]) →
fold (prepend, [8; 7; 6; 5], []) →
[8; 7; 6; 5]
```



## 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 list =
  fold ((fun (a, x) -> a + x), 0, list)
```

```
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'd)

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

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

```
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 add_n (n, list) =
  let add x = n + x in
  map (add, list)
```

Accessing variable from outer scope

– (equivalent to...)

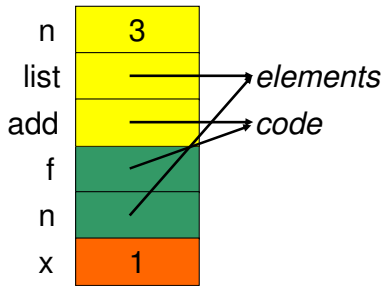
```
let add_n (n, list) =
  map ((fun x -> n + x), list)
```



# Consider the Call Stack Again

```
let map (f, n) = match n with
   [] -> []
| (h::t) -> (f h)::(map (f, t))
let add_n (n, list) =
  let add x = n + x in
  map (add, list)

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




- Uh oh...how does add know the value of n?
  - The **wrong** answer for OCaml: 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 uses add_n's n
  - C example:

```
int x = 1;

void f() {
  x++;
}

void g() {
  int x = 2;
  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

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

  - When the anonymous function is called, n isn't even on the stack any more!
    - The language needs some way to keep n around after add_n 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