CMSC330

Ocaml – closures and currying
Last time

- Ocaml features
  - Tuples
  - Recursion
  - Higher-order functions
This time

- Closures
- Currying

Law of Closure:

Objects grouped together are seen as a whole.

We tend to ignore gaps and complete contour lines. In the image above, there are no triangles or circles, but our minds fill in the missing information to create familiar shapes and images.
The **fold** function

• **Common pattern**
  – Iterate through list and apply function to each element, keeping track of partial results computed so far
    
    ```ocaml
    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)
```

- Can you build the **rev** 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]
```
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

```ml
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
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;
}
Consider the call stack again

• 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

```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])
```
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 Function

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

\[
\text{let addN } n = (\text{fun } x \rightarrow x + n) \\
(\text{addN } 3) \ 4 \ (* \text{ returns 7 } *)
\]

– When the anonymous function is called, \( n \) isn’t even on the stack any more!
  • We need some way to keep \( n \) around after \text{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

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

\[(\text{add 3}) \ 4 \rightarrow \langle \text{cl} \rangle \ 4 \rightarrow 3 + 4 \rightarrow 7\]
Example – closure 2

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

(mult_sum (3, 4)) 5 -> <cl> 5 -> 5 * 7 -> 35
```
Example – closure 3

```ocaml
let twice (n, y) = 
    let f x = x + n in
    f (f y)
```

twice (3, 4) → <cl> (<cl> 4)→ <cl> 7 → 10
Example – closure 4

let add x = (fun y -> (fun z -> x + y + z))

add( ) took 3 arguments?

(((add 1) 2) 3) → (((<cl> 2) 3) → (<cl> 3) → 1+2+3

fun y ->
(fun z ->
 x+y+z)

x = 1

fun z ->
x+y+z

x = 1
y = 2
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 Schönfinkeling or Fregging
Curried function 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)
```
Curried Functions in Ocaml (cont.)

• What is the type of add?

```ocaml
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

  – \( \rightarrow \) associates to the right
    • \( \text{int} \rightarrow \text{int} \rightarrow \text{int} \) is the same as \( \text{int} \rightarrow (\text{int} \rightarrow \text{int}) \)

  – Function application ( ) associates to the left
    • \( \text{add} \ 3 \ 4 \) is the same as \( (\text{add} \ 3) \ 4 \)
Another example of currying

- A curried add function with three arguments

  let add_th x y z = x + y + z

  is the same as

  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
Recall **map** and **fold**

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

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

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

  - **Type** = (`'a * 'b -> 'a) * 'a * 'b list -> 'a
Currying and the **map** function

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

- **Examples**

  ```ocaml
  let negate x = -x
  map negate [1; 2; 3] (* [-1; -2; -3] *)
  let negate_list = map negate
  negate_list [-1; -2; -3] (* [1; 2; 3] *)
  let sum_pair_l = map (fun (a, b) -> a + b)
  sum_pair_l[(1, 2); (3, 4)] (* [3; 7] *)
  ```

- **What is the type of this form of map?**

  ('a -> 'b) -> 'a list -> 'b list
Currying and the \textit{fold} function

- New Fold

\[
\text{let rec fold } f \ a \ l = \text{match } l \text{ with} \\
[] \rightarrow a \\
| (h::t) \rightarrow \text{fold } f \ (f \ a \ h) \ t
\]

- Examples

\[
\begin{align*}
\text{let add } x \ y &= x + y \\
\text{fold add } 0 \ [1; 2; 3] &= (* 6 *) \\
\text{let sum } &= \text{fold add } 0 \\
\text{sum } [1; 2; 3] &= (* 6 *) \\
\text{let next } n ~_&= n + 1 \\
\text{let len } &= \text{fold next } 0 \\
\text{len } [4; 5; 6; 7; 8] &= (* 5 *)
\end{align*}
\]

- What is the type of this form of \textit{fold}?

\[('a \rightarrow 'b \rightarrow 'a) \rightarrow 'a \rightarrow 'b \text{ list } \rightarrow 'a\]
Another convention

• Since functions are curried, `function` can often be used instead of `match`
  
  – `function` declares anonymous function w/ one argument
  
  – Instead of

    ```
    let rec sum l = match l with
    [] -> 0
    | (h::t) -> h + (sum t)
    ```

  
  – It could be written

    ```
    let rec sum = function
    [] -> 0
    | (h::t) -> h + (sum t)
    ```
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
Higher-Order Functions in C

• C supports function pointers

```c
typedef int (*int_func)(int);
void app(int_func f, int *a, int n) {
    for (int i = 0; i < n; i++)
        a[i] = f(a[i]);
}
int add_one(int x) { return x + 1; }
int main() {
    int a[] = {5, 6, 7};
    app(add_one, a, 3);
}
```
Higher-Order Functions in C (cont.)

- C does not support closures
  - Since no nested functions allowed
  - Unbound symbols always in global scope

```c
int y = 1;
void app(int(*f)(int), n) {
    return f(n);
}
int add_y(int x) {
    return x + y;
}
int main() {
    app(add_y, 2);
}
```
Higher-Order Functions in Ruby

• Ruby supports higher-order functions
  – Use `yield` within method to call code block argument

```ruby
def my_collect(a)
  b = Array.new(a.length)
  0.upto(a.length-1) { |i|
    b[i] = yield(a[i])
  }
  return b
end

b = my_collect([5, 6, 7]) { |x| x+1 }
```
Higher-Order Functions in Ruby (cont.)

- Ruby supports closures
  - Code blocks can access non-local variables
  - Binding determined by lexical scoping

```ruby
def twice
  yield
  yield
end
x = 1
twice {x += 1}
puts x  # 3
```

```ruby
def twice
  x = 0  #dynamic
  yield
  yield
end
x = 1  #lexical
twice {x += 1}
puts x  # 3 not 1
```
Higher-Order Functions in Ruby (cont.)

- Ruby code blocks are actual variables

```ruby
def twice  # implicit block
    yield  # invoked with yield
    yield
end
twice { x += 1 }  # same as x += 2

↓
def quad (&block)  # explicit block
    twice (&block)  # used as argument
twice (&block)
end
quad { x += 1 }  # same as x += 4
```
Higher-Order Functions in Ruby (cont.)

• Code blocks may be saved

```ruby
def quad (&block)  # explicit block
  c = block       # no ampersand!
  twice (c)       # used as argument
  twice (c)
end

↓

def twice c       # arg = explicit closure
  c.call         # invoke with .call
  c.call
end

quad { x += 1 }  # same as x += 4
```
Closures in Ruby

- Ruby supports creating closures directly
  - Proc.new
  - proc
  - lambda
  - method

```ruby
c1 = Proc.new { x+=1 }
c2 = proc       { x+=1 }
c3 = lambda     { x+=1 }
def foo
  x+=1
end
c4 = method    { :foo }
c.call         # x+=1
```
Higher-Order Functions in Java/C++

• An object in Java or C++ is kind of like a closure
  – It has some data (like an environment)
  – Along with some methods (i.e., function code)
  – So objects can be used to simulate closures

• So is an anonymous Java inner class
  – Inner class methods can access fields of outer class

• Back in CMSC 132 (OOP II)
  – We studied how to implement some functional patterns in OO languages
Summary

• Closures
• Currying