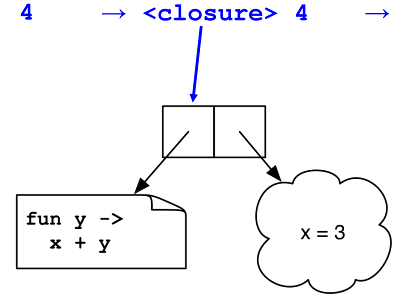
CMSC 330: Organization of Programming Languages

Functional Programming with OCaml, con't.

Example

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

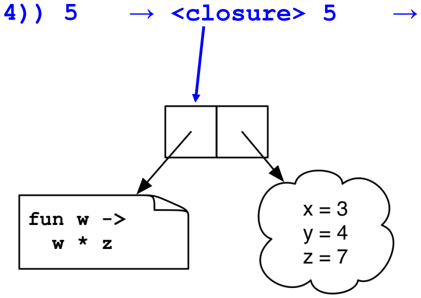
```
(add 3) 4    → <closure> 4    → 3 + 4    → 7
```



Another Example

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

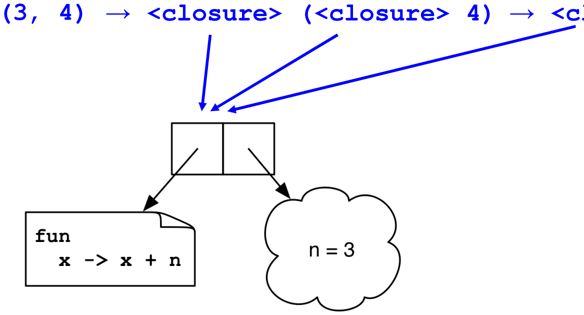
```
(mult_sum (3, 4)) 5    → <closure> 5    → 5 * 7    → 35
```



Yet Another Example

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

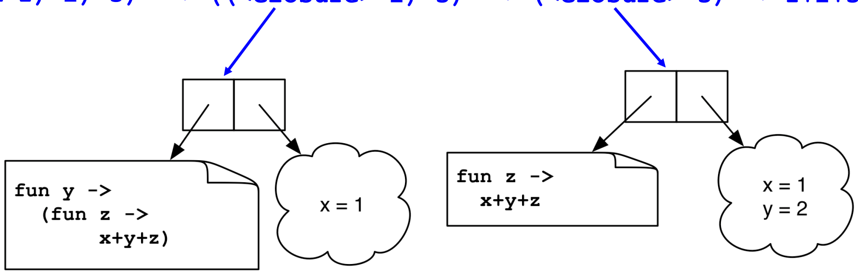
```
twice (3, 4) → <closure> (<closure> 4) → <closure> 7 → 10
```



Still Another Example

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

`(((add 1) 2) 3)` → `((<closure> 2) 3)` → `(<closure> 3)` → `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

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

  - This is identical to all of the following:

```
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'd)

- 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`

Another Example of Currying

- A curried add function with three arguments:

```
let add_three x y z = x + y + z
```

  – The same as

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

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

Currying and the map Function

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

- Examples

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

- What's the type of this form of **`map`**?

Currying and the fold Function

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

```
let add x y = x + y
fold add 0 [1; 2; 3]
let sum = fold add 0
sum [1; 2; 3]
let next n _ = n + 1
let length = fold next 0   (* warning:  not polymorphic *)
length [5; 6; 7; 8]
```

- What's the type of this form of **`fold`**?

Another Convention

- Since functions are curried, function can often be used instead of match, function declares an anonymous function of 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)
```

Another Convention (cont'd)

Instead of

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

It could be written

```
let rec map f = function
  [] -> []
| (h::t) -> (f h)::(map f 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 Grace
    - 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 has function pointers but not closures
  - (gcc has closures)

```
typedef int (*int_func)(int);

void app(int_func f, int *a, int n) {
  int i;
  for (i = 0; i < n; i++)
    a[i] = f(a[i]);
}

int add_one(int x) { return x + 1; }

int main() {
  int a[] = {1, 2, 3, 4};
  app(add_one, a, 4);
}
```


Higher-Order Functions in Ruby

- Use yield within a method to call a code block argument

```
def my_collect(a)
  b = Array.new(a.length)
  i = 0
  while i < a.length
    b[i] = yield(a[i])
    i = i + 1
  end
  return b
end

b = my_collect([1, 2, 3, 4, 5]) { |x| -x }
```

Higher-Order Functions in Java/C++

- An object in Java or C++ is kind of like a closure
  - it's some data (like an environment)
  - along with some methods (i.e., function code)

- So objects can be used to simulate closures

- Later we'll look at how to implement some functional programming patterns in OO languages