

CMSC 330: Organization of Programming Languages

OCaml – Recursion & Higher Order Functions

Lists and Recursion

- ▶ Lists have a recursive structure
 - And so most functions over lists will be recursive

```
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?

CMSC 330

3

Examples – Recursive Functions

- ▶ `last l` (* last element of l *)

```
let rec last l = match l with
  [x] -> x
  | (_::xs) -> last xs
```
- ▶ `append (l, m)`
(* 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))
```

CMSC 330

5

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

```
let rec print_up_to (n, m) =
  print_int n; print_string "\n";
  if n < m then print_up_to (n + 1, m)
```

CMSC 330

2

Examples – Recursive Functions

- ▶ `sum l` (* sum of elts in l *)

```
let rec sum l = match l with
  [] -> 0
  | (x::xs) -> x + (sum xs)
```
- ▶ `negate l` (* negate elements in list *)

```
let rec negate l = match l with
  [] -> []
  | (x::xs) -> (-x) :: (negate xs)
```

CMSC 330

4

Examples – Recursive Functions

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

```
let rec rev l = match l with
  [] -> []
  | (x::xs) -> append ((rev xs), [x])
```
- `rev` takes $O(n^2)$ time. Can you do better?

CMSC 330

6

A Clever Version of Reverse

```
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

```
rev [1; 2; 3] →
rev_helper ([1;2;3], []) →
rev_helper ([2;3], [1]) →
rev_helper ([3], [2;1]) →
rev_helper ([], [3;2;1]) →
[3;2;1]
```

CMSC 330

7

Examples – Recursive Functions

- `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))`

CMSC 330

8

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:

```
a = [1,2,3,4,5]
b = a.collect { |x| -x }
```

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

CMSC 330

9

Higher-Order Functions

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

```
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)
```

CMSC 330

10

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]) = [2; 3; 4]
map (negate, [9; -5; 0]) = [-9; 5; 0]
```

CMSC 330

11

The map Function (cont.)

- What is the type of the map function?

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

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

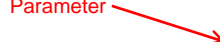

CMSC 330

12

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

Parameter  `fun x -> x + 3` 

```
twice ((fun x -> x + 3), 5) = 11
map ((fun x -> x+1), [1; 2; 3]) = [2; 3; 4]
```

CMSC 330

13

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]
```

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

```
map ((fun (x, y) -> x+y), [(1,2); (3,4)])
    = [3; 7]
```

CMSC 330

14

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 = 8
```

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

```
let f x = body
      ↓ stands for
let f = fun x -> body
```

CMSC 330

15

Examples – Anonymous Functions

- ▶ `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)
```

CMSC 330

16

Examples – Anonymous Functions

- ▶ `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))`

CMSC 330

17

The fold Function

- ▶ Common pattern

- Iterate through list and apply function to each element, keeping track of 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"

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

- ▶ What's the type of `fold`?

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

CMSC 330

18

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!

CMSC 330

19

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!

CMSC 330

20

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, [], [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]
```

CMSC 330

21

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;
}
```

x	4	f
x	3	g
y	4	
z	3	h

CMSC 330

22

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 *)
```

CMSC 330

23

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
```

CMSC 330

24

How About This?

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

Accessing variable
from outer scope

- (Equivalent to...)

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

CMSC 330

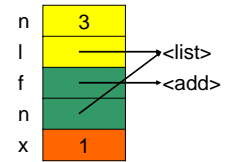
25

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])



- 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

CMSC 330

26

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:

Refers to the `x` at file scope – that's the nearest `x` going from inner scope to outer scope in the source code

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

CMSC 330

27