CMSC 631

Functional Programming with Ocaml

(last updated Sep 8, 2011)

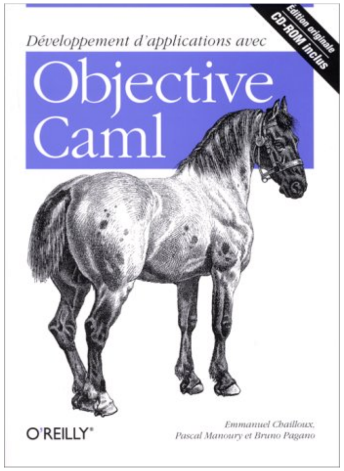
Background

- ML (Meta Language)
  - Univ. of Edinburgh, 1973
  - Part of a theorem proving system LCF
    - The Logic of Computable Functions
- SML/NJ (Standard ML of New Jersey)
  - Bell Labs and Princeton, 1990
  - Now Yale, AT&T Research, Univ. of Chicago (among others)
- OCaml (Objective CAML)
  - INRIA, 1996
  - French Nat'l Institute for Research in Computer Science

Dialects of ML

- Other dialects include MoscowML, ML Kit, Concurrent ML, Manticore, and others
  - But SML/NJ and OCaml are most popular
  - O = "Objective," but probably won't cover objects
  - Haskell is arguably an ML too, but for lazy evaluation
- Languages all have the same core ideas
  - But small and annoying syntactic differences
  - So you should not buy a book with ML in the title
    - Because it probably won't cover OCaml

More Information on OCaml



- Translation available on the class webpage
  - *Developing Applications with Objective Caml*
- Webpage also has link to another book
  - *Introduction to the Objective Caml Programming Language*

Features of ML

- Higher-order functions
  - Functions can be parameters and return values
- "Mostly functional"
- Data types and pattern matching
  - Convenient for certain kinds of data structures
- Type inference
  - No need to write types in the source language
    - But the language is statically typed
  - Supports *parametric polymorphism*
    - *Generics* in Java, *templates* in C++
- Exceptions
- Garbage collection

Functional languages

- In a pure functional language, every program is just an expression evaluation

let add1 x = x + 1;;

let rec add (x,y) = if x=0 then y else add(x-1, add1(y));;

add(2,3) = add(1,add1(3)) = add(0,add1(add1(3)))
= add1(add1(3)) = add1(3+1) = 3+1+1
= 5

OCaml has this basic behavior, but has additional features to ease the programming process.
- Less emphasis on data storage
- More emphasis on function execution

A Small OCaml Program- Things to Notice

Use (* *) for comments (may nest)

Use let to bind variables and *: type* to assign a type

```
(* A small OCaml program *)
let x : int = 37;;
let y : int = x + 5;;
Printf.printf
  "%d\n" y;;
```

Ocaml has printf

;; ends a top-level expression

Line breaks, spacing ignored (like C, C++, Java, not like Ruby)

Run, OCaml, Run

- OCaml programs can be compiled using ocamlc
  - Produces .cmo ("compiled object") and .cmi ("compiled interface") files
    - We'll talk about interface files later
  - By default, also links to produce executable a.out
    - Use -o to set output file name
    - Use -c to compile only to .cmo/.cmi and not to link
    - You'll be given a Makefile if you need to compile your files

Run, OCaml, Run (cont'd)

- Compiling and running the previous small program:

ocaml1.ml:

```
(* A small OCaml program *)
let x : int = 37;;
let y : int = x + 5;;
Printf.printf
  "%d\n" y;;
```

```
% ocamlc ocaml1.ml
% ./a.out
42
%
```


Run, OCaml, Run (cont'd)

Expressions can also be typed and evaluated at the top-level:

```
# 3 + 4;;
- : int = 7

# let x : int = 37;;
val x : int = 37

# x;;
- : int = 37

# let y : int = 5;;
val y : int = 5

# let z : int = 5 + x;;
val z : int = 42

# print_int z;;
42- : unit = ()

# print_string "Colorless green ideas sleep furiously";;
Colorless green ideas sleep furiously- : unit = ()

# print_int  "Colorless green ideas sleep furiously";;
This expression has type string but is here used with type int
```

gives type and value of each expr

"-" = "the expression you just typed"

unit = "no interesting value" (like void)

Run, OCaml, Run (cont'd)

- Files can be loaded at the top-level

```
% ocaml
        Objective Caml version 3.12.1

# #use "ocaml1.ml";;
val x : int = 37
val y : int = 42
42- : unit = ()

- : unit = ()
# x;;
- : int = 37
```

ocaml1.ml:

```
(* A small OCaml program *)
let x : int = 37;;
let y : int = x + 5;;
print_int y;;
print_string "\n";;
```

#use loads in a file one line at a time

Defining Functions

list parameters and their types after function name

use let to define functions

return type

```
let next (x:int) : int = x + 1;;
next 3;;
```

no parentheses on function calls

no return statement

Local Variables

- You can use let inside of functions for locals

```
let area (r:float):float =
  let pi:float = 3.14 in
  pi *. r *. r
```

  – And you can use as many lets as you want

```
let area (d:float):float =
  let pi:float = 3.14 in
  let r:float = d /. 2.0 in
  pi *. r *. r
```

Floating point multiplication operator is *. while for ints it's just *
Same goes with +. and +
In short: Ocaml does not support overloaded operators

More on the local let

- let x = e1 in e2 means
  – Evaluate e1, then
  – Evaluate e2, with x bound to result of evaluating e1
  – x is *not* visible outside of e2

```
let pi:float = 3.14 in pi *. 3.0 *. 3.0;;
pi;;
```

bind pi in body of let

floating point multiplication

error

More on the Let Construct (cont'd)

- Compare to similar usage in Java/C

```
let pi:float = 3.14 in
  pi *. 3.0 *. 3.0;;
pi;;
```

```
{
  float pi = 3.14;

  pi * 3.0 * 3.0;
}
pi;
```

- In the top-level, omitting in means "from now on":

  # let pi:float = 3.14;;
  (* pi is now bound in the rest of the top-level scope *)

Type Annotations

- The syntax **e:t** asserts that "**e** has type **t**"
  – This can be added anywhere you like

```
let x : int = 3
let z = (x : int) + 5
```

- Use to give functions parameter and return types

```
let fn (x:int):float =
        (float_of_int x) *. 3.14
```

  – Note special position for return type
    - Thus `let g x:int = ...` means `g` returns `int`

Type Inference

- All type annotations are optional
  - Ocaml will attempt to infer the types of bound variables if you do not specify them

```
let x : int = 3;;
let next (x:int):int = x + 1;;
```

can instead be written

```
let x = 3;;
let next x = x + 1;;
```

- Will go over how inference works later in course

Style guidelines for type inference

- Rule 1: Include type annotations on function arguments and return types for all but the most simple functions
- Rule 2: Leave off annotations on local variables unless initialized by a complicated expression
- Rationale:
  - Writing types helps document functionality
  - But adding type where the type is obvious (e.g., from seeing a local variable's initializer) does not help

Basic Types in OCaml

- Read e : t has "expression e has type t"

42 : int	true : bool
"hello" : string	'c' : char
3.14 : float	() : unit (* don't care value *)

- OCaml has static types to help you avoid errors
  - Note: Sometimes the messages are a bit confusing

```
# 1 + true;;
This expression has type bool but is here used with
  type int
```

  - Watch for the underline as a hint to what went wrong
  - But not always reliable
  - Use type annotations to help debug

Function Types

- In OCaml, `->` is the function type constructor
  - The type `t1 -> t2` is a function with argument or *domain* type `t1` and return or *range* type `t2`
- Examples
  - `let next (x:int):int = x + 1    (* int -> int *)`
  - `let fn (x:int):float = (float_of_int x) *. 3.14`
    `(* int -> float *)`
  - `print_string    (* string -> unit *)`
    - `unit` return type indicates the result is not of interest
  - Type a function name at top level to get its type

;; versus ;

- ;; ends an expression in the top-level of OCaml
  - Use it to say: "Give me the value of this expression"
  - Not used in the body of a function
  - Not needed after each function definition
- e1; e2 evaluates e1 and then e2, and returns e2
  - Typically, e1 has type `unit`

```
let print_both (s:string, t:string) =
  print_string s; print_string t; "Printed s and t."
```

  - notice no ; at end---it's a *separator*, not a *terminator*

```
print_both ("Colorless green ", "ideas sleep")
```

Prints `"Colorless green ideas sleep"`, and returns `"Printed s and t."`

Nested Let

- Uses of let can be nested

```
let pi = 3.14 in
let r = 3.0 in
  pi *. r *. r;;
(* pi, r no longer in scope *)
```

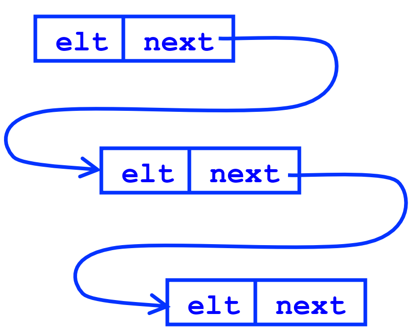
```
{
  float pi = 3.14;
  float r = 3.0;

  pi * r * r;
}
/* pi, r not in scope */
```


Lists in OCaml

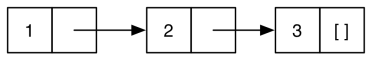
- The basic data structure in OCaml is the list
  - Lists are written as [e1; e2; ...; en]
    - # [1;2;3]
    - - : int list = [1;2;3]
  - Notice int list – lists must be *homogeneous*
  - The empty list is []
    - # []
    - - : 'a list
  - The 'a means "a list containing anything"
    - we'll see more about this later
  - Warning: Don't use a comma instead of a semicolon
    - Means something different (we'll see in a bit)

Consider a Linked List in C

```
struct list {
  int elt;
  struct list *next;
};
…
struct list *l;
…
i = 0;
while (l != NULL) {
  i++;
  l = l->next;
}
```

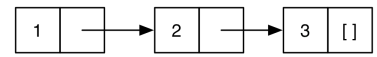


Lists in OCaml are Linked



- [1;2;3] is represented above
  - A nonempty list is a pair (element, rest of list)
  - The element is the *head* of the list
  - The pointer is the *tail* or *rest* of the list
    - ...which is itself a list!
- Thus in math a list is either
  - The empty list []
  - Or a pair consisting of an element and a list
    - This recursive structure will come in handy shortly

Lists are Linked (cont' d)



- :: prepends an element to a list
  - h::t is the list with h as the element at the beginning and t as the "rest"
  - :: is called a *constructor*, because it builds a list
  - Although it' s not emphasized, :: does allocate memory
- Examples

```
3::[]            (* The list [3] *)
2::(3::[])       (* The list [2; 3] *)
1::(2::(3::[]))  (* The list [1; 2; 3] *)
```


More Examples

```
# let y : int list = [1;2;3] ;;
val y : int list = [1; 2; 3]
# let x : int list = 4::y ;;
val x : int list = [4; 1; 2; 3]
# let z : int list = 5::y ;;
val z : int list = [5; 1; 2; 3]
```

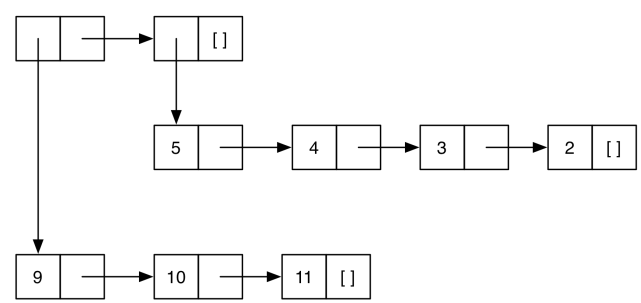
- *not* modifying existing lists, just creating new lists

```
# let w : int list list = [1;2]::y ;;
This expression has type int list but is here
  used with type int list list
```

- The left argument of :: is an element
- Can you construct a list y such that [1;2]::y makes sense?

Lists of Lists

- Lists can be nested arbitrarily
  - Example: `[ [9; 10; 11]; [5; 4; 3; 2] ]`
    - (Type `int list list`)



Pattern Matching

- To pull lists apart, use the match construct

  match e with p1 -> e1 | ... | pn -> en

- **p1**...**pn** are *patterns* made up of [], ::, and *pattern variables*
- **match** finds the first **pk** that matches the shape of **e**
  - Then **ek** is evaluated and returned
  - During evaluation of **pk**, pattern variables in pk are bound to the corresponding parts of **e**
- An underscore _ is a wildcard pattern
  - Matches anything
  - Doesn't add any bindings
  - Useful when you want to know something matches, but don't care what its value is

Example

match e with p1 -> e1 | ... | pn -> en

```
let is_empty (l:int list):bool = match l with
    [] -> true
  | (h::t) -> false

  is_empty []          (* evaluates to true *)
  is_empty [1]         (* evaluates to false *)
  is_empty [1;2;3]     (* evaluates to false *)
```


Pattern Matching (cont'd)

- `let hd l = match l with (h::t) -> h`
  - `hd [1;2;3]  (* evaluates to 1 *)`
- `let hd l = match l with (h::_) -> h`
  - `hd []       (* error!  no pattern matches *)`
- `let tl l = match l with (h::t) -> t`
  - `tl [1;2;3]  (* evaluates to [2; 3] *)`

Missing Cases

- Exceptions for inputs that don't match any pattern
  - OCaml will warn you about non-exhaustive matches
- Example:

```
# let hd l = match l with (h::_) -> h;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]

# hd [];;
Exception: Match_failure ("", 1, 11).
```

More Examples

- `let f l =`
  ```
  match l with (h1::(h2::_)) -> h1 + h2
  ```
  - `f [1;2;3]`
  - `(* evaluates to 3 *)`

- `let g l =`
  ```
  match l with [h1; h2] -> h1 + h2
  ```
  - `g [1; 2]`
  - `(* evaluates to 3 *)`
  - `g [1; 2; 3]`
  - `(* error! no pattern matches *)`

Abbreviated pattern matching

- `let f p = e`, where `p` is a pattern, is a shorthand for `let f x = match x with p -> e`
- Examples
  - `let hd (h::_) = h`
  - `let tl (_::t) = t`
  - `let f (x::y::_) = x + y`
  - `let g [x; y] = x + y`
- Type inference uses the pattern to infer the type
- Useful if there's only one acceptable input

Tuples

- Tuples are *constructed* using `(e1, ..., en)`
  - They're like C structs but without field labels, and allocated on the heap
  - Unlike lists, tuples do *not* need to be homogenous
  - E.g., `(1, ["string1"; "string2"])` is a valid tuple
- Tuples are *deconstructed* using pattern matching

Examples with Tuples

- `let plusThree (x,y,z) = x + y + z`
  `let addOne (x, y, z) = (x+1, y+1, z+1)`
  - `plusThree (addOne (3, 4, 5))` `(* returns 15 *)`
- `let sum ((a, b), c) = (a+c, b+c)`
  - `sum ((1, 2), 3)` `(* returns (4, 5) *)`
- `let plusFirstTwo (x::y::_, a) = (x + a, y + a)`
  - `plusFirstTwo ([1; 2; 3], 4)` `(* returns (5, 6) *)`
- `let tls (_::xs, _::ys) = (xs, ys)`
  - `tls ([1; 2; 3], [4; 5; 6; 7])` `(* returns ([2; 3], [5; 6; 7]) *)`
- Remember, semicolon for lists, comma for tuples
  - `[1, 2] = [(1, 2)] = a list of size one`
  - `(1; 2) = a syntax error`

Tuples Are a Fixed Size

```
# let foo x:? = match x with
  (a, b) -> a + b
| (a, b, c) -> a + b + c;;
This pattern matches values of type 'a * 'b * 'c
but is here used to match values of type 'd * 'e
```

- Thus there's never more than one match case with tuples

Tuple Types

- Tuple types use `*` to separate components

- Examples
  - `(1, 2) :`
  - `(1, "string", 3.5) :`
  - `(1, ["a"; "b"], 'c') :`
  - `[(1,2)] :`
  - `[(1, 2); (3, 4)] :`
  - `[(1,2); (1,2,3)] :`

Tuple Types

- Tuple types use `*` to separate components

- Examples
  - `(1, 2) : int * int`
  - `(1, "string", 3.5) : int * string * float`
  - `(1, ["a"; "b"], 'c') : int * string list * char`
  - `[(1,2)] : (int * int) list`
  - `[(1, 2); (3, 4)] : (int * int) list`
  - `[(1,2); (1,2,3)] : error`

Ocaml functions take one argument

- Consider this example

```
let plus (x, y) = x + y;;
plus (3, 4);;
```

  - It looks like you're passing in two arguments
  - Actually, you're passing in a *tuple* instead
    - And using abbreviated pattern matching – (x,y) is a tuple pattern!
- Annotation applies to single tuple argument

  `let plus (x,y:int*int):int = x + y;;`

  Not `let plus(x:int,y:int):int = x + y`

Lists and tuples

- `let f (l:int list):? =`
  `    match l with x::(_::y) -> (x,y)`

- What is `f [1;2;3;4]`?
  Possibilities:
  ```
  ([1],[3])
  (1,3)
  (1,[3])
  (1,4)
  (1,[3;4])
  ```


Type declarations

- type can be used to create new names for types
  - useful for combinations of lists and tuples

- Examples

```
type my_type = int * (int list)
(3, [1; 2]) : my_type

type my_type2 = int * char * (int * float)
(3, 'a', (5, 3.0)) : my_type2
```


Polymorphic Types

- If annotating the types of hd, we might write
  - `let hd (l:int list):int = match l with (h::t) -> h`
  - `hd [1; 2; 3]        (* returns 1 *)`
- Seems weird to write the same code for
  - `let hds (l:string list):string =`
    `match l with (h::t) -> h`
  - `hds ["a"; "b"; "c"](* returns "a" *)`
- *Polymorphic* types to the rescue
  - `let hd (l:'a list):'a = match l with (h::t) -> h`
    - Has type `'a list -> 'a`
  - this says the function takes a list of any element type `'a`, and returns something of that type

Examples of Polymorphic Types

- `let swap (x:'a,y:'b ):'b*'a = (y, x)`
  - `swap : 'a*'b -> 'b*'a`

- `let id (x:'a):'a = x`
  - `id: 'a -> 'a`

(Remember the type annotations are optional. Ocaml will infer exactly the same types if you remove them.)

Conditionals

- Use if...then...else just like C/Java
  – No parentheses and no end

```
if grade >= 90 then
  print_string "You got an A"
else if grade >= 80 then
  print_string "You got a B"
else if grade >= 70 then
  print_string "You got a C"
else
  print_string "You're not doing so well"
```


Conditionals (cont'd)

- In OCaml, conditionals return a result
  – The value of whichever branch is true/false
  – Like ? : in C, C++, and Java

```
# if 7 > 42 then "hello" else "goodbye";;
- : string = "goodbye"
# let x = if true then 3 else 4;;
x : int = 3
# if false then 3 else 3.0;;
This expression has type float but is here used
  with type int
```

- Putting this together with what we've seen earlier, can you write fact, the factorial function?

The Factorial Function

```
let rec fact (n:int):int =
  if n = 0 then
     1
  else
     n * fact (n-1);;
```

- Notice no return statements
  – So this is pretty much how it needs to be written
- The rec part means "define a recursive function"
  – This is special for technical reasons
  – let x = e1 in e2   x in scope within e2
  – let rec x = e1 in e2   x in scope within e2 *and e1*
    - OCaml will complain if you use let instead of let rec

More examples of let

- `(let x = 1 in x) ; x;;`
- `let x = x in x;;`
- `let x = 4;;`
  `let x = x + 1 in x;;`
- `let f n = 10;;`
  `let f n = if n = 0 then 1 else n * f (n – 1);;`
  `f 0;;`
  `f 1;;`
- `let f x = f x;;`

More examples of let

- `(let x = 1 in x) ; x;;  (* error, x is unbound *)`
- `let x = x in x;;  (* error, x is unbound *)`
- `let x = 4;;`
  `let x = x + 1 in x;;  (* 5 *)`
- `let f n = 10;;`
  `let f n = if n = 0 then 1 else n * f (n – 1);;`
  `f 0;; (* 1 *)`
  `f 1;; (* 10 *)`
- `let f x = f x;;  (* error, f is unbound *)`

Recursion = Looping

- Recursion is essentially the only way to iterate
  – (The only way we're going to talk about)
- Another example

```
let rec print_up_to (n:int, m:int):unit =
  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

```
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?

More Examples

- `sum l (* sum of elts in l *)`

```
let rec sum (l:int list):int = match l with
    [] -> 0
  | (x::xs) -> x + (sum xs)
```

- `neg l (* negate elements in list *)`

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

- `last l  (* last element of l *)`

```
let rec last (l:'a list):'a = match l with
    [x] -> x
  | (x::xs) -> last xs
```

More Examples (cont' d)

```
(* return a list containing all the elements in the
   list l followed by all the elements in list m *)
```

- `append (l, m)`

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

- `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?

A Clever Version of Reverse

```
let rec rev_helper((l:'a list),(a:'a list)):'a
  list =
match l with
    [] -> a
  | (x::xs) -> rev_helper (xs, (x::a))
let rev (l:'a list):'a list = 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]
```


More Examples

- `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 elts of l *)`

```
let rec take (n, l) =
  if n = 0 then []
  else match l with
      [] -> []
    | (x::xs) -> x :: (take (n-1, xs))
```


Pattern Matching Lists of Lists

- You can do pattern matching on these as well
- Examples
  - `let addFirsts ((x::_) :: (y::_) :: _) = x + y`
    - `addFirsts [ [1; 2; 3]; [4; 5]; [7; 8; 9] ] = 5`
  - `let addFirstSecond ((x::_)::(_::y::_)::_) = x + y`
    - `addFirstSecond [ [1; 2; 3]; [4; 5]; [7; 8; 9] ] = 6`
- Note: You probably won't do this much or at all
  - You'll mostly write recursive functions over lists
  - We'll see that soon

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)
  (* ('a->'a) * 'a  -> 'a *)
twice (plus_three, 5)

let plus_four x = x + 4
let pick_fn n = (* int -> (int->int) *)
    if n > 0 then plus_three else plus_four
(pick_fn 5) 0
```


The map Function

- Let's write the map function
  - 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 often we don't want to bother to give them names
- Use fun to make a function with no name

Parameter   Body

```
fun x -> x + 3
```

```
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))`

The fold Function

- Common pattern: iterate through a list and apply a function to each element, keeping track 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, [], [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.

```
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

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’d)

- 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
```


How About This?

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

Accessing variable from outer scope

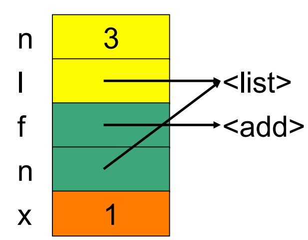
– (Equivalent to...)

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


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?
  – 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 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(); }
```

Returned Functions

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

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

Environments and Closures

- An *environment* is a mapping from variable names to values
  - Just like a stack frame

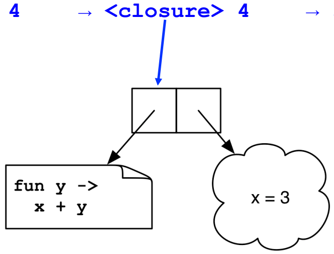
- A *closure* is a pair (f, e) consisting of function code f and an environment e

- When you invoke a closure, f is evaluated using e to look up variable bindings

Example

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

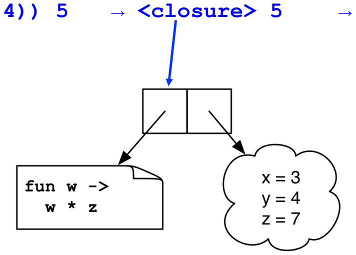
(add 3) 4 → <closure> 4 → 3 + 4 → 7



Another Example

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

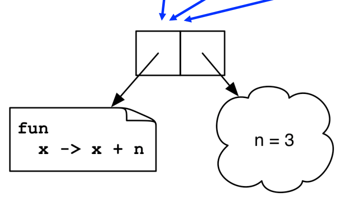
(mult_sum (3, 4)) 5 → <closure> 5 → 5 * 7 → 35



Yet Another Example

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

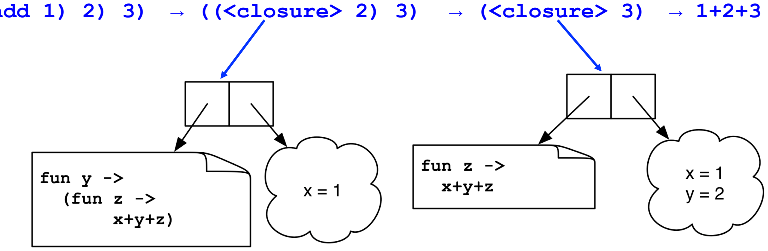
```
twice (3, 4) → <closure> (<closure> 4) → <closure> 7  → 10
```



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_th x y z = x + y + z
```

  - 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`

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 [4; 5; 6; 7]
```

- 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/lib/ocaml on junkfood
    - 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 no closures
  - (gcc had 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 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
- When we get to Java in the course, we'll study how to implement some functional patterns in OO languages

OCaml Data

- So far, we've seen the following kinds of data:
  - Basic types (int, float, char, string)
  - Lists
    - One kind of data structure
    - A list is either [] or h::t, deconstructed with pattern matching
  - Tuples
    - Let you collect data together in fixed-size pieces
  - Functions
- How can we build other data structures?
  - Building everything from lists and tuples is awkward

Data Types

```
type shape =
    Rect of float * float    (* width * length *)
  | Circle of float          (* radius *)

let area s =
  match s with
      Rect (w, l) -> w *. l
    | Circle r -> r  *. r *. 3.14

area (Rect (3.0, 4.0))
area (Circle 3.0)
```

- **`Rect`** and **`Circle`** are *type constructors*- here a **`shape`** is either a **`Rect`** or a **`Circle`**
- Use pattern matching to *deconstruct* values, and do different things depending on constructor

Data Types (cont'd)

```
type shape =
    Rect of float * float    (* width * length *)
  | Circle of float

let l = [Rect (3.0, 4.0) ; Circle 3.0; Rect (10.0, 22.5)]
```

- What's the type of `l`?
- What's the type of `l`'s first element?

Data Types (cont'd)

- The *arity* of a constructor is the number of arguments it takes
  – A constructor with no arguments is *nullary*

```
type optional_int =
   None
 | Some of int

let add_with_default a = function
     None -> a + 42
   | Some n -> a + n

add_with_default 3 None       (* 45 *)
add_with_default 3 (Some 4)   (* 7 *)
```

  – Constructors must begin with uppercase letter

Polymorphic Data Types

```
type 'a option =
    None
  | Some of 'a

let add_with_default a = function
    None -> a + 42
  | Some n -> a + n

add_with_default 3 None       (* 45 *)
add_with_default 3 (Some 4)   (* 7 *)
```

- This option type can work with any kind of data
  – In fact, this option type is built-in to OCaml

Recursive Data Types

- Do you get the feeling we can build up lists this way?

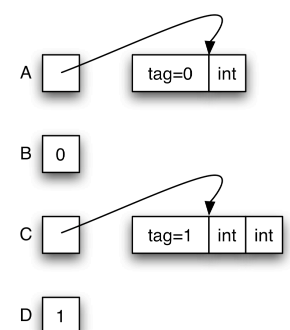
```
type 'a list =
   Nil
 | Cons of 'a * 'a list

let rec length l = function
   Nil -> 0
 | Cons (_, t) -> 1 + (length t)

length (Cons (10, Cons (20, Cons (30, Nil))))
```

  – Note: Don't have nice [1; 2; 3] syntax for this kind of list

Data Type Representations

- Values in a data type are stored either directly as integers or as pointers to blocks in the heap

```
type t =
  A of int
| B
| C of int * int
| D
```



Exercise: A Binary Tree Data Type

- Write type **`bin_tree`** for binary trees over **`int`**
  - trees should be ordered

- Implement the following

```
empty : bin_tree
is_empty : bin_tree -> bool
member : int -> bin_tree -> bool
insert : int -> bin_tree -> bin_tree
remove: int -> bin_tree -> bin_tree
equal : bin_tree -> bin_tree -> bool
fold : (int -> 'a -> 'a) -> bin_tree -> 'a -> 'a
```


Exceptions

```
exception My_exception of int

let f n =
  if n > 0 then
    raise (My_exception n)
  else
    raise (Failure "foo")

let bar n =
  try
    f n
  with My_exception n ->
      Printf.printf "Caught %d\n" n
    | Failure s ->
      Printf.printf "Caught %s\n" s
```


Exceptions (cont’d)

- Exceptions are declared with exception
  - They may appear in the signature as well
- Exceptions may take arguments
  - Just like type constructors
  - May also be nullary
- Catch exceptions with try...with...
  - Pattern-matching can be used in with
  - If an exception is uncaught
    - Current function exits immediately
    - Control transfers up the call chain
    - Until the exception is caught, or until it reaches the top level

Modules

- So far, most everything we’ve defined has been at the “top-level” of OCaml
  - This is not good software engineering practice

- A better idea: Use *modules* to group associated types, functions, and data together
  - Avoid polluting the top-level with unnecessary stuff

- For lots of sample modules, see the OCaml standard library

Creating a Module

```
module Shapes =
  struct
    type shape =
       Rect of float * float    (* width * length *)
     | Circle of float          (* radius *)

    let area = function
       Rect (w, l) -> w *. l
     | Circle r -> r  *. r *. 3.14

    let unit_circle = Circle 1.0
  end;;

unit_circle;;   (* not defined *)
Shapes.unit_circle;;
Shapes.area (Shapes.Rect (3.0, 4.0));;
open Shapes;;   (* import all names into current scope *)
unit_circle;;   (* now defined *)
```


Modularity and Abstraction

- Another reason for creating a module is so we can *hide* details
  - Ex: Binary tree module
    - May not want to expose exact representation of binary trees
  - This is also good software engineering practice
    - Prevents clients from relying on details that may change
    - Hides unimportant information
    - Promotes local understanding (clients can't inject arbitrary data structures, only ones our functions create)

Module Signatures

Entry in signature

Supply function types

```
module type FOO =
  sig
    val add : int -> int -> int
  end;;

module Foo : FOO =
  struct
    let add x y = x + y
    let mult x y = x * y
  end;;

Foo.add 3 4;;     (* OK *)
Foo.mult 3 4;;    (* not accessible *)
```

Give type to module

Module Signatures (cont'd)

- Convention: Signature names in all-caps
  - This isn't a strict requirement, though
- Items can be omitted from a module signature
  - This provides the ability to hide values
- The default signature for a module hides nothing
  - You'll notice this is what OCaml gives you if you just type in a module with no signature at the top-level

Abstract Types in Signatures

```
module type SHAPES =
  sig
    type shape
    val area : shape -> float
    val unit_circle : shape
    val make_circle : float -> shape
    val make_rect : float -> float -> shape
end;;

module Shapes : SHAPES =
  struct
    ...
    let make_circle r = Circle r
    let make_rect x y = Rect (x, y)
  end
```

- Now definition of **`shape`** is hidden

Abstract Types in Signatures

```
# Shapes.unit_circle
- : Shapes.shape = <abstr>  (* OCaml won't show impl *)
# Shapes.Circle 1.0
Unbound Constructor Shapes.Circle
# Shapes.area (Shapes.make_circle 3.0)
- : float = 29.5788
# open Shapes;;
# (* doesn't make anything abstract accessible *)
```

- How does this compare to modularity in...
  - C?
  - C++?
  - Java?

.ml and .mli files

- Put the signature in a foo.mli file, the struct in a foo.ml file
  - Use the same names
  - Omit the sig...end and struct...end parts
  - The OCaml compiler will make a Foo module from these

Example

shapes.mli

```
type shape
val area : shape -> float
val unit_circle : shape
val make_circle : float -> shape
val make_rect : float -> float -> shape
```

shapes.ml

```
type shape =
  Rect of ...
...
let make_circle r = Circle r
let make_rect x y = Rect (x, y)
```

```
% ocamlc shapes.mli   # produces shapes.cmi
% ocamlc shapes.ml    # produces shapes.cmo
ocaml
# #load "shapes.cmo"  (* load Shapes module *)
```

Functors

- Modules can take other modules as arguments
  - Such a module is called a *functor*
  - You're mostly on your own if you want to use these
- Example: **`Set`** in standard library

```
module type OrderedType = sig
  type t
  val compare : t -> t -> int
end

module Make(Ord: OrderedType) =
struct ... end

module StringSet = Set.Make(String);;
(* works because String has type t,
implements compare *)
```


So Far, only Functional Programming

- We haven't given you *any* way so far to change something in memory
  - All you can do is create new values from old
- This actually makes programming *easier* !
  - Don't care whether data is shared in memory
    - Aliasing is irrelevant
  - Provides strong support for compositional reasoning and abstraction
    - Ex: Calling a function f with argument x always produces the same result

Imperative OCaml

- There are three basic operations on memory:
  - **`ref : 'a -> 'a ref`**
    - Allocate an updatable reference
  - **`! : 'a ref -> 'a`**
    - Read the value stored in reference
  - **`:= : 'a ref -> 'a -> unit`**
    - Write to a reference

```
let x = ref 3  (* x : int ref *)
let y = !x
x := 4
```


Comparison to L- and R-values

- Recall that in C/C++/Java, there's a strong distinction between l- and r-values
  - An *r-value* refers to just a value, like an integer
  - An *l-value* refers to a location that can be written

l-value

y = x;

r-value

- A variable's meaning depends on where it appears
  - On the right-hand side, it's an r-value, and it refers to the contents of the variable
  - On the left-hand side of an assignment, it's an l-value, and it refers to the location the variable is stored in

L-Values and R-Values (cont'd)

Store 3 in location x

```
int x, y;

x = 3;

y = x;

3 = x;
```

Read contents of x and store in location y

Makes no sense

- Notice that x, y, and 3 all have type **`int`**

Comparison to OCaml

```
int x, y;

x = 3;

y = x;

3 = x;
```

```
let x = ref 0;;
let y = ref 0;;

x := 3;;   (* x : int ref *)

y := (!x);;

3 := x;;   (* 3 : int; error *)
```

- In OCaml, an updatable location and the contents of the location have different types
  - The location has a **`ref`** type

Capturing a ref in a Closure

- We can use refs to make things like counters that produce a fresh number "everywhere"

```
let next =
  let count = ref 0 in
    function () ->
      let temp = !count in
        count := (!count) + 1;
        temp;;

# next ();;
- : int = 0
# next ();;
- : int = 1
```


Semicolon Revisited; Side Effects

- Now that we can update memory, we have a real use for ; and () : unit
  - e1; e2 means evaluate e1, throw away the result, and then evaluate e2, and return the value of e2
  - () means "no interesting result here"
  - It's only interesting to throw away values or use () if computation does something besides return a result
- A *side effect* is a visible state change
  - Modifying memory
  - Printing to output
  - Writing to disk

Grouping with begin...end

- If you're not sure about the scoping rules, use begin...end to group together statements with semicolons

```
let x = ref 0

let f () =
  begin
    print_string "hello";
    x := (!x) + 1
  end
```


The Trade-Off of Side Effects

- Side effects are absolutely necessary
  - That's usually why we run software! We want something to happen that we can observe
- They also make reasoning harder
  - Order of evaluation now matters
  - Calling the same function in different places may produce different results
  - Aliasing is an issue
    - If we call a function with refs r1 and r2, it might do strange things if r1 and r2 are aliased

OCaml Language Choices

- Implicit or explicit declarations?
  - Explicit – variables must be introduced with let before use
  - But you don't need to specify types
- Static or dynamic types?
  - Static – but you don't need to state types
  - OCaml does *type inference* to figure out types for you
  - Good: less work to write programs
  - Bad: easier to make mistakes, harder to find errors

OCaml Programming Tips

- Compile your program often, after small changes
  - The OCaml parser often produces inscrutable error messages; it's easier to figure out what's wrong if you've only changed a few things since the last compile
- If you're getting strange type error messages, add in type declarations
  - Try writing down types of arguments
  - And for any expression e, can write (e:t) to assert e has type t

OCaml Programming Tips (cont'd)

- Watch out for operator precedence and function application

```
let mult x y = x*y

mult 2 2+3    (* returns 7 *)
              (* parsed as (mult 2 2)+3 *)

mult 2 (2+3)  (* returns 10 *)
```


OCaml Programming Tips (cont'd)

- All branches of a pattern match must return the same type

```
match x with
... -> -1    (* branch returns int *)
| ... -> ()   (* uh-oh, branch returns unit *)
| ... -> print_string "foo"
              (* also returns unit *)
```


OCaml Programming Tips (cont'd)

- You cannot assign to ordinary variables!

```
# let x = 42;;
val x : int = 42
# x = x + 1;;        (* this is a comparison *)
-: bool = false
# x := 3;;
Error: This expression has type int but is here
used with type 'a ref
```


OCaml Programming Tips (cont'd)

- Again: You cannot assign to ordinary variables!

```
# let x = 42;;
val x : int = 42
# let f y = y + x;;    (* captures x = 42*)
val f : int -> int = <fun>
# let x = 0;;          (* shadows binding of x *)
val x : int = 0
# f 10;;               (* but f still refers to x=42 *)
- : int = 52
```