CMSC 631
Fall 2006

Functional Programming with OCaml

Background
- 1973 – ML developed at Univ. of Edinburgh
  - Part of a theorem proving system LCF
    - The Logic of Computable Functions
- SML/NJ (“Standard ML of New Jersey”)
  - http://www.smlnj.org
  - Developed at Bell Labs and Princeton; now Yale, AT&T Research, Univ. of Chicago (among others)
- OCaml
  - http://www.ocaml.org
  - Developed at INRIA (The French National Institute for Research in Computer Science)

Dialects of ML
- Other dialects include MoscowML, ML Kit, Concurrent ML, etc.
  - But SML/NJ and OCaml are most popular
  - O = “Objective,” but probably won’t cover objects
- 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 course web site
  - Developing Applications with Objective Caml
- Web site 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
- Static Type System
- Type inference
  - No need to write types in the source language
  - Supports parametric polymorphism (generics in Java, templates in C++)
- Exceptions
- Garbage Collection

Basic OCaml
- OCaml has a read-eval-print loop for the top level

```
(* This is an OCaml program *)
let x = 37;;
let y = x + 5;;
print_int y;;
print_string "\n";;
```

```
% ocamlc ocaml1.ml
% ./a.out
42
% ```
Things to Notice

- Use (* *) for comments (may nest)
- Use let to bind variables
- No type declarations
- Need to use correct print function (OCaml also has printf)
- **;** ends a top-level expression

**OCaml**

```
let x = 37;;
let y = x + 5;;
print_int y;;
print_string "\n";;
```

- Use (* *) for comments (may nest)
- Ending a top-level expression with **;**

Run, OCaml, Run

- Can compile OCaml programs 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** and not to link
  - We’ll give you a Makefile if you need to compile your files

Run, OCaml, Run (cont’d)

- Alternatively, use the OCaml top-level directly
  - % ocaml
    ```
    Objective Caml version 3.08.3
    # use "ocaml1.ml";;
    val x : int = 37
    val y : int = 42
    42- : unit = ()
    # use loads in a file one line at a time
    # Gives type and value of each expr
    Unit = "no interesting value" like void
    "-" = "the expression you just typed"
    ```

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
  - Watch for the underline as a hint to what went wrong
  - But not always reliable

More on the Let Construct

- **let** is more often used for local variables
  - 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
  - Error
    ```
    let pi = 3.14 in
    pi * 3.0 * 3.0;;
    pi;
    ```

More on the Let Construct (cont’d)

- Compare to similar usage in Java/C
  ```
  let pi = 3.14 in
  pi * 3.0 * 3.0;
  pi;
  ```

- In the top-level, omitting **in** means “from now on”:
  ```
  # let pi = 3.14;;
  (* pi is now bound in the rest of the top-level scope *)
  ```
Nested Let

- Uses of `let` can be nested

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

```ocaml
{ float pi = 3.14;
  float r = 3.0;
  pi * r * r;
} /* pi, r not in scope */
```

Defining Functions

- Use `let` to define functions

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

```ocaml
let plus (x, y) = x + y;;
plus (3.0, 4.0);;
```

Local Variables

- You can use `let` inside of functions for locals

```ocaml
let area r =
  let pi = 3.14 in
  pi *. r *. r
```

```ocaml
let area d =
  let pi = 3.14 in
  let r = d /. 2.0 in
  pi *. r *. r
```

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 = x + 1 (* type int -> int *)`
  - `let foo x = (float_of_int x) *. 3.14 (* type int -> float *)`
  - `print_string (* type string -> unit *)`

- Type in fn name at top level to get type

Type Annotations

- Syntax `(e : t)` to assert "e has type t"
  - You can add this anywhere you like
    ```ocaml
    let (x : int) = 3
    let z = (x : int) + 5
    ```
  - Use to give functions param and return types
    ```ocaml
    let foo (x:int):float =
      (float_of_int x) *. 3.14
    ```
  - Note special position for return type
  - Thus `let bar x:int = ...` means `bar` returns `int`
  - Very useful for debugging
  - Especially for more complicated types

`;;` versus `;

- `;;` ends an expression in the top-level of OCaml
  - Use it to say: "Give me the value of this expression"
  - You’ll never use this in the body of a function
  - You don’t need to add it after each function defn
    - Though for now it won’t hurt if you do

- `e1; e2` evaluates `e1` and then `e2`, and returns `e2`
  - `let print_both (s, t) = print_string s; print_string t;;`
    - Notice, no ; at end–it’s a separator, not a terminator
  - `print_both ("hello", "goodbye");;`
    - Prints hello<newline>goodbye<newline>
  - Returns "goodbye"
Lists in OCaml

- The basic data structure in OCaml is the list
  - Write down a list as \([e_1; e_2; \ldots; e_n]\)
    
    # [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 much more about this later
  - Warning: Don’t use comma instead of semicolon
    - Means something different (we’ll see in a bit)

Lists are Linked

\[
\begin{array}{c}
1 \\
2 \\
3 \\
i
\end{array}
\]

- \([1;2;3]\) is represented above
  - A non-empty 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)

\[
\begin{array}{c}
1 \\
2 \\
3 \\
i
\end{array}
\]

- :: prepends an element to a list
  - h::t is the list with h as the element at 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 = [1;2;3] ;;
  
  val y : int list = [1; 2; 3]
- let x = 4::y ;;
  
  val x : int list = [4; 1; 2; 3]
- let z = 5::y ;;
  
  val z : int list = [5; 1; 2; 3]
  - Not modifying existing lists, just creating new lists
- let w = [1;2]::y ;;
  
  This expression has type int list but is here used with type int list list
    - The left argument to :: 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 a with p1 -> e1 | ... | pn -> en

- let is_empty l = 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 on the match Construct

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

• p1...pn are patterns
  – Made up of [], ::, and pattern variables
  – Can use _ (underscore) when don’t care about value

• Match finds the first pattern pk that matches the shape of e

• Then the pattern variables in pk are bound to the corresponding parts of e while ek is evaluated and returned

More Examples

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

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

An Abbreviation

• Can write let f p = e (* p is a pattern *)
  – Shorthand for let f x = match x with p -> e

• Examples
  - let hd (h::_) = h
  - let tl (_,::t) = t
  - let foo (x::y:::_ ) = x + y
  - let bar (x: y) = x + y

• Useful if there’s only one acceptable input

OCaml Functions Take One Argument

• Recall 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 pattern matching

• Tuples are constructed using (e1, ... , en)
  – They’re like C structs but no labels; alloc’d on heap
  – Unlike lists, 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) = (4, 5)

- let plusFirstTwo (x::y::_, a) = (x + a, y + a)
- plusFirstTwo ([1; 2; 3], 4) = (5, 6)

- let tls (_::xs::__; ys) = (xs, ys)
- tls ([1; 2; 3], [4; 5; 6; 7]) = ([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

List and Tuple Types

- Tuple types are built using * to separate sub-components

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

Type declarations

- Can use type to create new names for types
  - Useful for combinations of lists and tuples

  - Examples
    - type my_type = int * (int list)
    - type my_type2 = int * char * (int * float)

Examples of Polymorphic Types

- let tl _::t = t
  - tl : 'a list -> 'a list

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

- let tls (_::xs, __::ys) = (xs, ys)
  - tls : 'a list * 'b list -> 'a list * 'b list

Polymorphic Types

- Some fns we saw require specific list types
  - let plusFirstTwo (x::y::_, a) = (x + a, y + a)
    - plusFirstTwo : int list * int -> (int * int)

- But other fns work for any list
  - let hd h:_ = h
    - hd : 'a list -> 'a

OCaml gives such fns polymorphic types

- hd : 'a list -> 'a
  - Takes a list of any element type 'a, and return something of that type

Examples of Polymorphic Types

- 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 *)

Polymorphic Types

- Some fns we saw require specific list types
  - let plusFirstTwo (x::y::_, a) = (x + a, y + a)
    - plusFirstTwo : int list * int -> (int * int)

- But other fns work for any list
  - let hd h:_ = h
    - hd : 'a list -> 'a

OCaml gives such fns polymorphic types

- hd : 'a list -> 'a
  - Takes a list of any element type 'a, and return something of that type

Tuples Are a Fixed Size

- Thus never >1 match case with tuples
Conditionals

- Use `if...then...else` just like C/Java
  - No parens, no end

```ocaml
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

```ocaml
# if ? > 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

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

- Notice: No return statements
  - So this is pretty much how we need to write it
- The `rec` part means “define a recursive function”
  - This is special for technical reasons
    - `let` in scope within `e2`
    - `let rec` in scope within `e2` and `e1`
- OCaml will complain if you use `let` instead of `let rec`

Recursion = Looping

- Recursion is essentially the only way to iterate
  - (The only way we’re going to tell you about)
- Another example

```ocaml
let rec print_up_to (n, m) =
  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

```ocaml
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 non-empty list is 1 plus the length of the tail
- Type of length?
  - `length : 'a list -> int`

More Examples

- `sum l (* sum of elts in l *)`
  ```ocaml
  let rec sum l = match l with
    [] -> 0
  | (x::xs) -> x + (sum xs)
  ```
- `negate l (* negate elements in list *)`
  ```ocaml
  let rec negate l = match l with
    [] -> []
  | (x::xs) -> (-x) :: (negate xs)
  ```
More Examples (cont’d)

• append (l, m) (* return l followed by 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) = match l with
| [] -> a
| (x::xs) -> rev_helper (xs, (x::a))

let rev l = rev_helper (l, [])

Higher-Order Functions

• In OCaml, 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) (* returns 11 *)
twice : ('a->'a) * 'a  ->  'a

let plus_four x = x + 4
let pick_one n = if n > 0 then plus_three else plus_four
  (pick_one 5) 0  (* returns 3 *)
pick_one : int -> (int->int)

The map Function

• Let’s write the map function
  – Takes a list and a function, and applies the function to
each element in the list
  let rec map (f, l) = match l with
  | [] -> []
  | (h::t) -> (f h)::(map (f, t))
  (* map : ('a -> 'b) * 'a list -> 'b list *)

let add_one x = x + 1
let negate x = -x
map (add_one, [1; 2; 3]) (* returns [2; 3; 4] *)
map (negate, [9; -5; 0]) (* returns [-9; 5; 0] *)

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

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

Pattern Matching with fun

• Can use match within fun

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

  – For complicated matches, though, use named functions

• Can use standard pattern matching abbreviation

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 short-hand
  - 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 keep track of something at the same time

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

(* fold : (a list a -> a) * a * a list -> a *)
```

- a = "accumulator"
- This is usually called "fold left" to remind us that f takes the accumulator as its first argument

Example

```
let add (a, x) = a + x
fold (add, 0, [1; 2; 3; 4])
fold (add, 1, [2; 3; 4])
fold (add, 2, [3; 4])
fold (add, 3, [4])
fold (add, 4, [])
```

We just built the `sum` function!

Another Example

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

let next (a, _) = a + 1
fold (next, 0, [1; 2; 3; 4])
fold (next, 1, [2; 3; 4])
fold (next, 2, [3; 4])
fold (next, 3, [4])
fold (next, 4, [])
```

We just built the `length` function!

Using fold to Build `rev`

- Can you build the `rev` 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], [])
```

[10; 9; 8; 7]
Nested Functions
- In OCaml, you can define functions anywhere
  - Even inside of other functions

  ```ocaml
  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

  ```ocaml
  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?
- (Equivalent to...)

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

Static Scoping
- In static or lexical scoping, 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(); }
    ```

  Refer to `x` at file scope – that’s nearest 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

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

    - When anonymous function is called, `n` isn’t even on the stack any more!
    - 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 → \(<\text{closure}>\) 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 → \(<\text{closure}>\) 5 → 5 * 7 → 35

Yet Another Example

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

\(foo\) (3, 4) → \(<\text{closure}>\) (\(<\text{closure}>\) 4) → \(<\text{closure}>\) 7 → 10

Still Another Example

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

\(add\) 1 2 3 → \(<\text{closure}>\) 2 3 → \(<\text{closure}>\) 3 → \(<\text{closure}>\) 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 logician Haskell B. Curry
  - But Schönfinkel and Frege discovered it
  - Should probably be called Schönfinkelization

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

- Thus:
  - \(\text{add}\) has type \(\text{int} \rightarrow (\text{int} \rightarrow \text{int})\)
  - \(add\ 3\) has type \(\text{int} \rightarrow \text{int}\)
  - \(\text{add}\ 3\) is a function that adds 3 to its argument
  - \(\text{add}\ 3\) \(\rightarrow\) 4 = 7

- Works for any number of arguments
Curried Functions in OCaml (cont’d)

• Because currying is so common, OCaml uses the following conventions:
  • \( \rightarrow \) associates to the right
    – Thus \( \text{int} \rightarrow \text{int} \rightarrow \text{int} \) is the same as
      \( \text{int} \rightarrow (\text{int} \rightarrow \text{int}) \)
  • application associates to the left
    – Thus \( \text{add} \, 3 \, 4 \) is the same as
      \( (\text{add} \, 3) \, 4 \)

Another Example of Currying

• A curried add fn with three args:
  ```ocaml
  let add_th x y z = x + y + z
  ```
  – same as
  ```ocaml
  let add_th x = (fun y -> (fun z -> x+y+z))
  ```
  • Then...
    – \( \text{add} \, \text{th} \) has type \( \text{int} \rightarrow (\text{int} \rightarrow \text{int}) \)
    – \( \text{add} \, \text{th} \, 4 \) has type \( \text{int} \rightarrow (\text{int} \rightarrow \text{int}) \)
    – \( \text{add} \, \text{th} \, 4 \, 5 \) has type \( \text{int} \rightarrow \text{int} \)
    – \( \text{add} \, \text{th} \, 4 \, 5 \, 6 \) is 15

Currying and the map Function

```ocaml
let rec map f l = match l with
  [] -> []
| (h::t) -> (f h)::(map f t) (* map : ('a -> 'b) -> 'a list -> 'b list *)
```

• Examples
  ```ocaml
  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 (function (a, b) -> a+b)
  sum_pairs_list [(1, 2); (3, 4)] (* [3; 7] *)
```

Currying and the fold Function

```ocaml
let rec fold f a l = match l with
  [] -> a
| (x::xs) -> fold f (f a x) xs

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]
```

Another Convention

• Since functions are curried, can often use function instead of match
  – Note: function declares anon func of one arg
  – Instead of
    ```ocaml
    let rec sum l = match l with
      [] -> 0
      | (x::xs) -> x + (sum xs)
    ```
    – Can write
    ```ocaml
    let rec sum = function
      [] -> 0
      | (x::xs) -> x + (sum xs)
    ```

Another Convention (cont’d)

Instead of
```ocaml
let rec map f l = match l with
  [] -> []
| (h::t) -> (f h)::(map f t)
```
Can write
```ocaml
let rec map f = function
  [] -> []
| (h::t) -> (f h)::(map f t)
```
Currying is Standard in OCaml

- Pretty much all functions are curried
  - Like standard library `map`, `fold`, etc.
  - In particular, look at the file `list.ml` for standard list funcs
  - 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
  - Unnecessary much of the time
  - Since functions are usually called with all arguments

OCaml Data

- So far, we’ve seen the following kinds of data:
  - Basic types (int, float, char, string)
  - Tuples
    - Lets you collect data together in fixed-size pieces
  - Functions
  - Lists
    - One kind of data structure
    - A list is either `[]` or `h::t` deconstructed with pattern matching

- How can we build other data structures?
  - Building everything from lists and tuples is clunky

Data Types

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

let area = function
  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)

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

```
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 *)
```

- (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

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

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

length (Cons [1, Cons [2, Cons [3, 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

\[
\text{type t} = \\
\text{A of int} \\
\text{B} \\
\text{C of int * int} \\
\text{D}
\]

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 OCaml standard library

Creating a Module

```ocaml
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 *. 3.14 *. 3.14
  let unit_circle = Circle 1.0
end;

unit_circle;; (* Not defined *)
Shapes.unit_circle;;
Shapes.area (Shapes.Rect (3.0, 4.0));;
unit_circle;; (* Now defined *)
```

Modularity and Abstraction

- Another reason for creating a module is so that we can hide details
  - For example, we built a binary tree module, but we may not want to expose our 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

```ocaml
module type FOO = 
sig
  val add : int -> int -> int
end;;
module Foo : FOO = 
struct
  let add x y = x + y
 ...
end;;
```

Module Signatures (cont’d)

- Convention is for signatures to be all caps
  - No strict requirements, though
- Can omit items from module signature
  - Provides ability to hide values
- Default signature for 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

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

.ml and .mli files

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

Example

shapes.ml
  type shape
  val area : shape -> float
  val unit_circle : shape
  val make_circle : float -> shape
  val make_rect : float -> float -> shape

% ocamlc shapes.mli # produces shapes.cmi
% ocamlc shapes.ml # produces shapes.cmo

ocaml
  # load "shapes.cmo" (* load in 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

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**

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

```ocaml
let x = ref 3
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 l-value refers to a location that can be written
  - An r-value refers to just a value, like an integer
- A var’s meaning depends on where it appears
  - On the left-hand side of an assignment, it’s an l-value, and it refers to the location the variable is stored in
  - On the right-hand side, it’s an r-value, and it refers to the contents of the variable

**L-Values and R-Values (cont’d)**

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

**Comparison to OCaml**

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

```ocaml
let x = ref 0
let y = ref 0
x := 3
y := (!x)
3 := x
```

**Capturing a ref in a Closure**

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

```ocaml
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`
  - `()` 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
  - Writing to 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

```ocaml
let x = ref 0
let foo () = 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`, might do strange things if `r1` and `r2` are aliased

Exceptions

```ocaml
exception My_exception of int
let foo n = if n > 0 then raise (My_exception n) else raise (Failure "foo")
let bar n = try foo n with My_exception n -> Printf.printf "Caught %d\n" n | Failure s -> Printf.printf "Caught %s\n" s
```

Exceptions (cont’d)

- Exceptions declared with `exception`
  - May appear in signature as well

- Exceptions may take arguments
  - Just like type constructors
  - May also be nullary

- Catch exceptions with `try...with...
  - Can use pattern matching in `with`
  - If an exception is uncaught, the current function exits immediately and control transfers up the call chain until the exception is caught, or until it reaches the top level