CMSC 330: Organization of Programming Languages

OCaml and Functional Programming

Dialects of ML

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

Dialects of ML (cont.)

- Other dialects
  - MoscowML, ML Kit, Concurrent ML, etc…
  - SML/NJ and OCaml are most popular

- 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

Features of ML

- “Mostly functional”
  - Some assignments
- Higher-order functions
  - Functions can be parameters and return values
- 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++

Features of ML (cont.)

- Data types and pattern matching
  - Convenient for certain kinds of data structures
- 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
```

Functional Languages (cont.)

- OCaml has similar basic behavior
  - Program execution = expression evaluation
- But has additional features
  - To ease the programming process
  - Features support
    - Less emphasis on data storage
    - More emphasis on function execution

A Small OCaml Program – Things to Notice

Use let to bind variables

Use (* *) for comments (may nest)

No type declarations

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

Need to use correct print function (OCaml also has printf)

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

;; ends a top-level expression

OCaml Interpreter

Expressions can be typed and evaluated at the top-level

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

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

# x;;
- : int = 37

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

# let z = 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)

OCaml Interpreter (cont.)

- Files can be loaded at top level

```
% ocaml
        Objective Caml version 3.08.3

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

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

#use loads in a file one line at a time

ocaml1.ml

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

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 can use a Makefile if you need to compile your files
- Not needed for this course

Run, OCaml, Run (cont.)

- Compiling & running the previous small program

ocaml1.ml

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

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

Basic Types in OCaml

- Read e : t as "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

The Let Construct

- let is often used for defining 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

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

error

bind pi in body of let

floating point multiplication

The Let Construct (cont.)

- Compare to similar usage in Java/C

```
let pi = 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 = 3.14;;

  (* pi is now bound in the rest of the top-level scope *)

;; 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
    - Though for now it won't hurt if used there
- e1; e2 evaluates e1 and then e2, and returns e2

```
let p (s,t) = print_int s; print_int t; "Done!"
```

  - Notice no ; at end
  - ; is a separator, not a terminator
  - Invoking `p (1,2)`
    - Prints `"1 2"`
    - Returns `"Done!"`

Examples – Semicolon

- Definition
  - e1 ; e2 (* evaluate e1, evaluate e2, return e2)
- 1 ; 2 ;;
  - (* 2 – value of 2^{nd} expression is returned *)
- (1 + 2) ; 4 ;;
  - (* 4 – value of 2^{nd} expression is returned *)
- 1 + (2 ; 4) ;;
  - (* 5 – value of 2^{nd} expression is returned to 1 + *)
- 1 + 2 ; 4 ;;
  - (* 4 – because + has higher precedence than ; *)

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 */
```

Defining Functions

Use let to define functions

List parameters after function name

```
let next x = x + 1;;
next 3;;
let plus (x, y) = x + y;;
plus (3, 4);;
```

No parentheses on function calls

No return statement

Local Variables

- You can use let inside of functions for locals

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

- And you can use as many lets as you want

```
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 fn x = (float_of_int x) *. 3.14
                             (* type int -> float *)
• print_string               (* type string -> unit *)
```

- Type a function name at top level to get its type

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 type of list is **`int list`**
    - Lists must be homogeneous

Lists in OCaml (cont.)

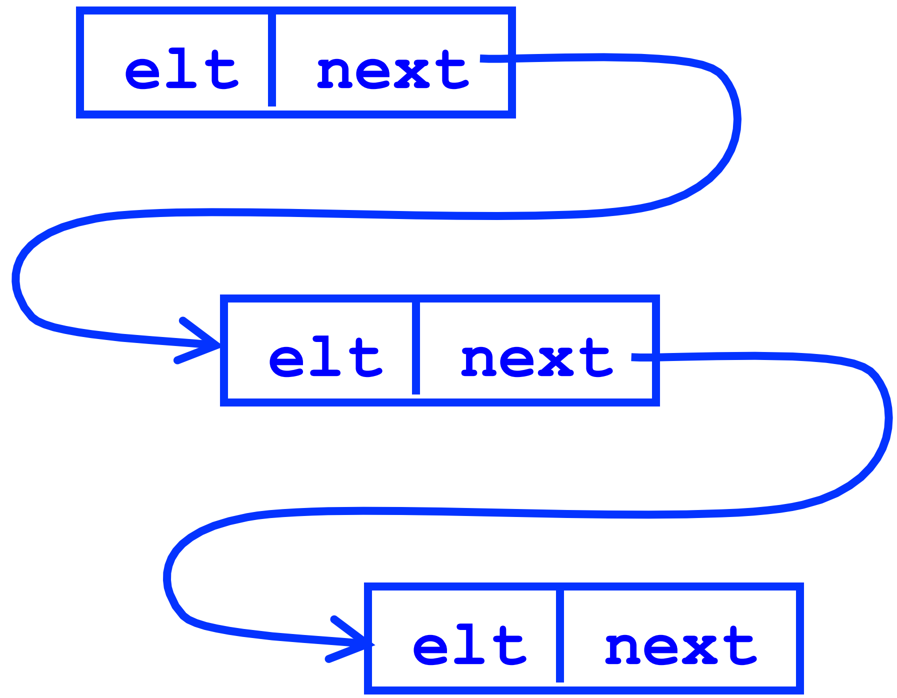
- More on OCaml lists
  - The empty list is [ ]

    ```
    # [ ]
    - : 'a list
    ```

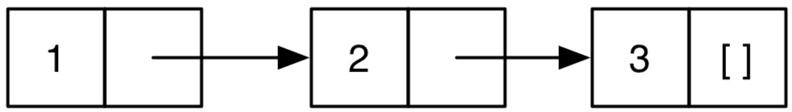
  - The `'a` means "a list containing anything"
    - We'll find out more about this later
  - List elements are separated using semicolons
  - Warning: Don't use a comma instead of a semicolon
    - Means something different (we'll see in a bit)

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;
}
```

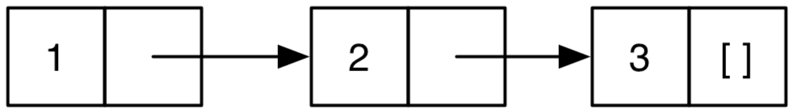


Lists in OCaml are Linked



- [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.)



- :: 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 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]
```

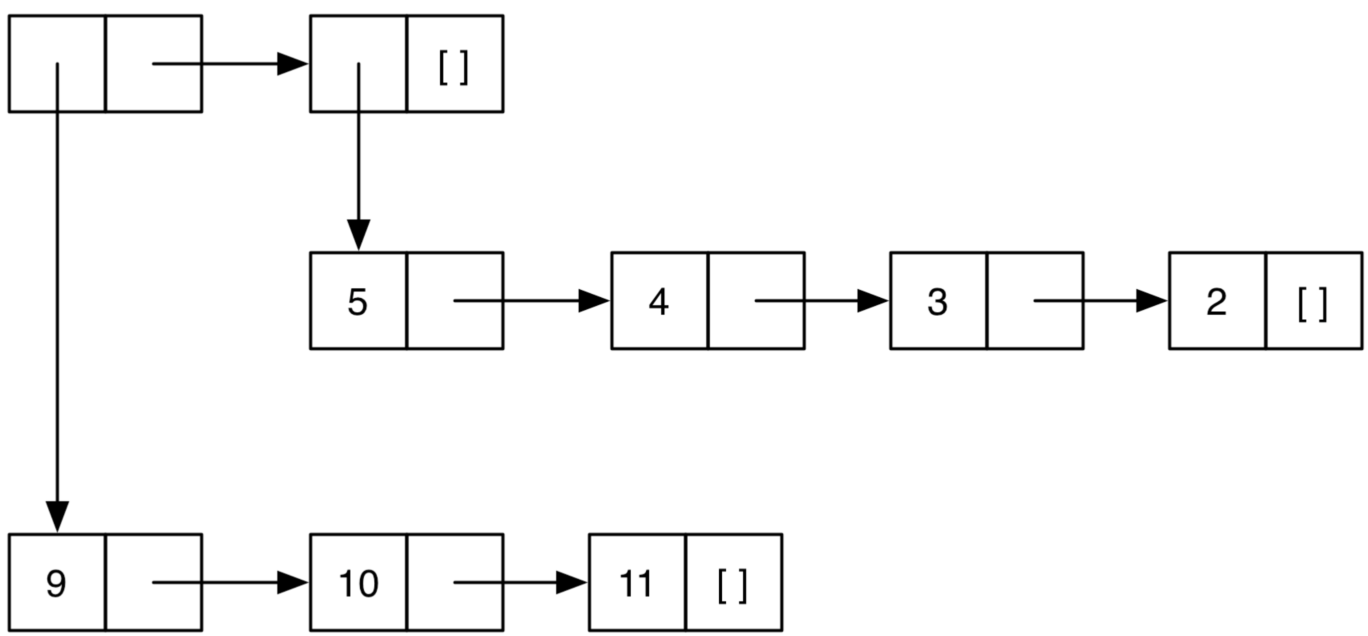
- not modifying existing lists, just creating new lists

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

- The left argument 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



Practice

- What is the type of
  - [1;2;3] `int list`
  - [[[ ]; [ ]; [1.3;2.4]]] `float list list list`
  - let func x = x::(0::[ ]) `int -> int 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 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`**

Pattern Matching Example

- Match syntax
  - `match e with p1 -> e1 | ... | pn -> en`
- Code 1
  - ```
    let is_empty l = match l with
            [] -> true
            | (h::t) -> false
    ```
- Outputs
  - `is_empty []    (* evaluates to true  *)`
  - `is_empty [1]   (* evaluates to false *)`
  - `is_empty [1;2](* evaluates to false *)`
    ```

Pattern Matching Example (cont.)

- Code 2
  - `let hd l = match l with (h::t) -> h`
- Outputs
  - `hd [1;2;3](* evaluates to 1 *)`
  - `hd [1;2]   (* evaluates to 1 *)`
  - `hd [1]     (* evaluates to 1 *)`
  - `hd []      (* Exception: Match failure *)`

Pattern Matching Example (cont.)

- Code 3
  - `let tl l = match l with (h::t) -> t`
- Outputs
  - `tl [1;2;3](* evaluates to [2;3] *)`
  - `tl [1;2]   (* evaluates to [2] *)`
  - `tl [1]     (* evaluates to [ ] *)`
  - `tl []      (* Exception: Match failure *)`

Pattern Matching – Wildcards

- 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

- In previous examples
  - Many values of `h` or `t` ignored
  - Can replace with wildcard _
  - Code behavior is identical

Pattern Matching – Wildcards (cont.)

- Code using _
  - `let is_empty l = match l with`
    `[] -> true | (_::_) -> false`
  - `let hd l = match l with (h::_) -> h`
  - `let tl l = match l with (_::t) -> t`
- Outputs
  - `is_empty[1](* evaluates to false *)`
  - `is_empty[ ](* evaluates to true *)`
  - `hd [1;2;3] (* evaluates to 1 *)`
  - `tl [1;2;3] (* evaluates to [2;3] *)`
  - `hd [1] (* evaluates to 1 *)`
  - `tl [1] (* evaluates to [ ] *)`

Pattern Matching – Missing Cases

- When pattern is defined
  - OCaml will warn you about non-exhaustive matches
- When pattern is used
  - Exceptions for inputs that don’t match any pattern
- 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).
```

Pattern Matching – An Abbreviation

- **`let f p = e`**, where p is a pattern
  - is 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`**
- Useful if there's only one acceptable input

Pattern Matching – Lists of Lists

- Can pattern match on lists of lists as well
- Examples
  - `let addFirsts`
    `((x::_) :: (y::_) :: _) = x + y`
    `addFirsts [[1;2];[4;5];[7;8;9]] =` 5
  - `let addFirstSecond`
    `(x::_)::(_::y::_)::_) = x + y`
    `addFirstSecond [[1;2];[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 instead

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

```
let plus t = match t with
    (x, y) -> x + y;;
plus (3, 4);;
```

  - And using pattern matching to extract its contents

Tuples

- Constructed using `(e1, ..., en)`
- Deconstructed using pattern matching
- Tuples are like C structs
  - But without field labels
  - Allocated on the heap
- Tuples can be heterogenous
  - Unlike lists, which must be homogenous
  - `(1, ["string1"; "string2"])` is a valid tuple

Tuples – Examples

- `let plusThree (x, y, z) = x+y+z`
  `let addOne (x, y, z) = (x+1, y+1, z+1)`
  - `plusThree (addOne (3,4,5)) =` 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)

Tuples – More Examples

- `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
- Example
  - `[1, 2] = [(1, 2)]` = a list of size one
  - `(1; 2)` = a syntax error

Another Tuple Example

- Given
  - `let f l = match l with x::(_::y) -> (x,y)`
- What is the value of
  - `f [1;2;3;4]`
- Possibilities
  - `([1],[3])`
  - `(1,3)`
  - `(1,[3])`
  - `(1,4)`
  - `(1,[3;4])`

List and 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`

Polymorphic Functions

- Some functions require specific list types
  - `let plusFirstTwo (x::y::_, a) =`
    `(x + a, y + a)`
  - `plusFirstTwo :`
    `int list * int -> (int * int)`
- But other functions work for a list of any type
  - `let hd (h::_) = h`
  - `hd [1; 2; 3]           (* returns 1   *)`
  - `hd ["a"; "b"; "c"]     (* returns "a" *)`
- These functions are polymorphic

Polymorphic Types

- OCaml gives such functions polymorphic types
  - `hd : 'a list -> 'a`
  - Read as
    - Function takes a list of any element type 'a
    - And returns something of that type
- Example
  - ```
    let tl (_::t) = t
    tl : 'a list -> 'a list
    ```
    ```

Polymorphic Types (cont.)

- More Examples
  - ```
    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
    ```

Tuples Are a Fixed Size

- This OCaml definition
  - `# let foo x = match x with`
    ```
    (a, b) -> a + b
    | (a, b, c) -> a + b + c;;
    ```
- Would yield this error message
  - This pattern matches values of type 'a * 'b * 'c but is here used to match values of type 'd * 'e
- Tuples of different size have different types
  - Thus never more than one match case with tuples

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

User Defined Types

- 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`

Data Types

- type can also be used to create variant types
  - Equivalent to C-style unions

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

- **`Rect`** and **`Circle`** are value constructors
  - Here a **`shape`** is either a **`Rect`** or a **`Circle`**

Data Types (cont.)

```
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)
```

- Use pattern matching to deconstruct values
  - **s** is a **shape**
  - Do different things for **s** depending on its constructor

Data Types (cont.)

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

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

- What's the type of `l`?

  `shape list`

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

  `shape`

Data Types Constructor

- Constructors must begin with uppercase letter
- 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
```

- Example
  - Arity of `None` = 0
  - Arity of `Some` = 1

Polymorphic Data Types

```
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  *)
```

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

Recursive Data Types

- We can build up lists this way

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

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

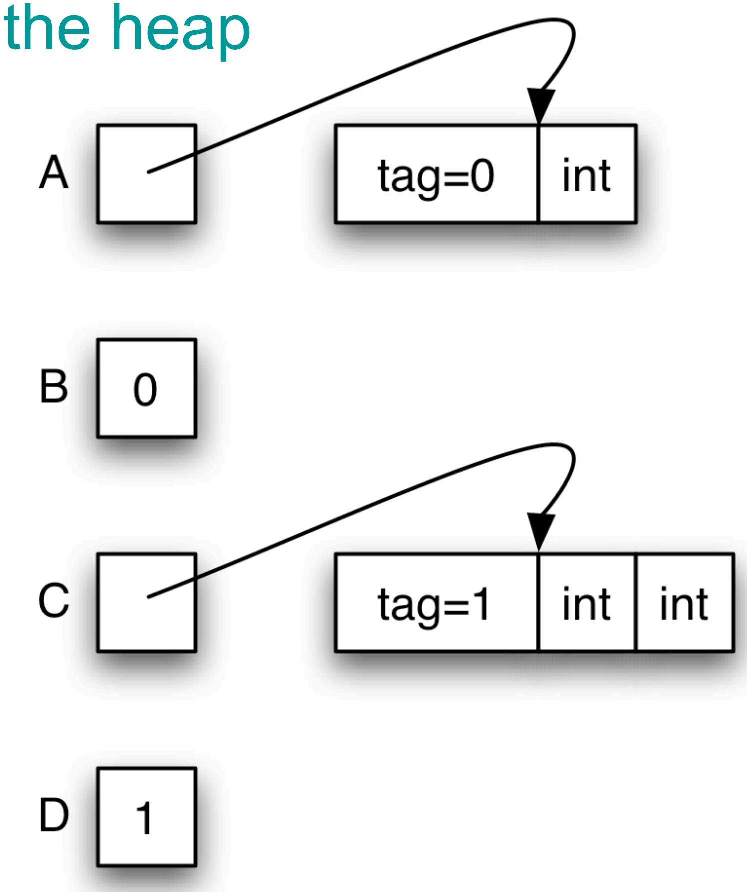
len (Cons (10, Cons (20, Cons (30, Nil))))
```

  - Won't have nice [1; 2; 3] syntax for this kind of list

Data Type Representations

- Values in a data type are stored
  1. Directly as integers
  2. 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 (binary search tree)
- 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
```

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`
- Not needed for this course
- But can be useful for debugging
  - Especially for more complicated types

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.)

- 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
```

The Factorial Function

- Using conditionals & functions
  - Can you write fact, the factorial function?

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

- Notice no return statements
  - This is pretty much how it needs to be written

Let Rec

- The rec part means "define a recursive function"
- Let vs. let rec
  - let x = e1 in e2 x in scope within e2
  - let rec x = e1 in e2 x in scope within e2 and e1
- Why use let rec?
  - If you used let instead of let rec to define fact

```
let fact n =
  if n = 0 then 1
  else n * fact (n-1) in e2
```

Fact is not bound here!

Examples – Let

- x;;
  - (* Unbound value x *)
- let x = 1 in x + 1;;
  - (* 2 *)
- let x = x in x + 1;;
  - (* Unbound value x *)

Examples – Let

- let x = 1 in(x + 1 ; x) ;;
  - (* 1 – ; has higher precedence than let … in *)
- (let x = 1 in x + 1) ; x;;
  - (* Unbound value x *)
- let x = 4 in(let x = x + 1 in x);;
  - (* 5 *)

Let – More Examples

- 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 … in … f …
  - (* Unbound value f *)
- let rec f x = … f … in … f …
  - (* Bound value f *)

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)
```

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?

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)
```

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))
```

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?

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]
```

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))
```

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?

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)
```

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])
map (negate, [9; -5; 0])
```

```
= [2; 3; 4]
= [-9; 5; 0]
```

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))
```

$$\underbrace{\texttt{('a -> 'b)}}_{\texttt{f}}\ \texttt{*}\ \underbrace{\texttt{'a list}}_{\texttt{l}}\ \texttt{-> 'b list}$$

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
```

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

- 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]
```

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
```

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)
```

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))
    ```
    ```

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
```

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.)

- 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

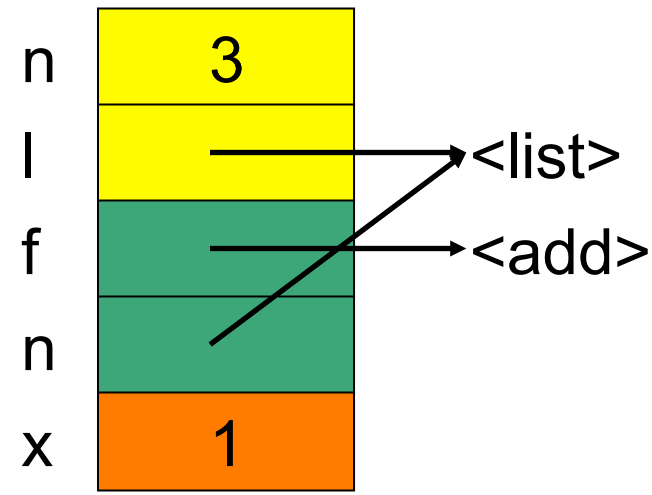
- (Equivalent to...)

```
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?
  - 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

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

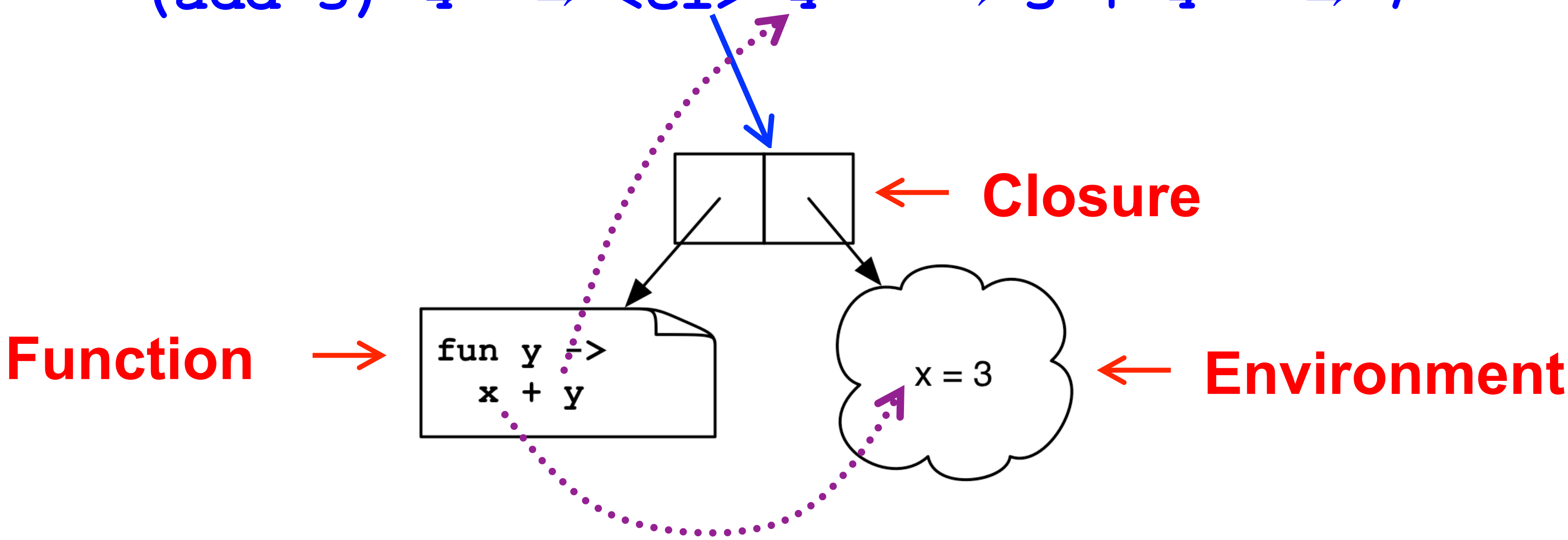
Environments and Closures

- 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 – Closure 1

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

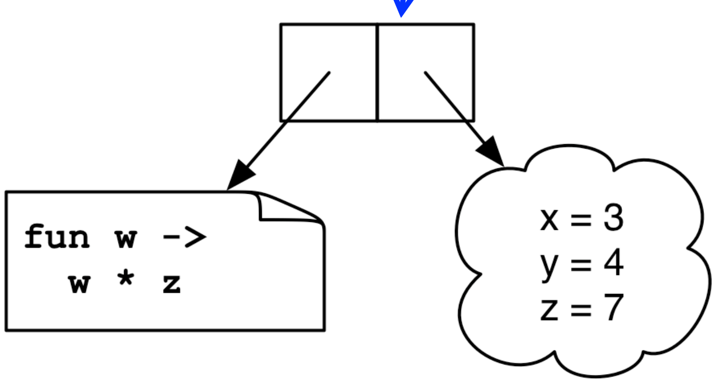
```
(add 3) 4   →  <cl> 4    →  3 + 4   →  7
```



Example – Closure 2

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

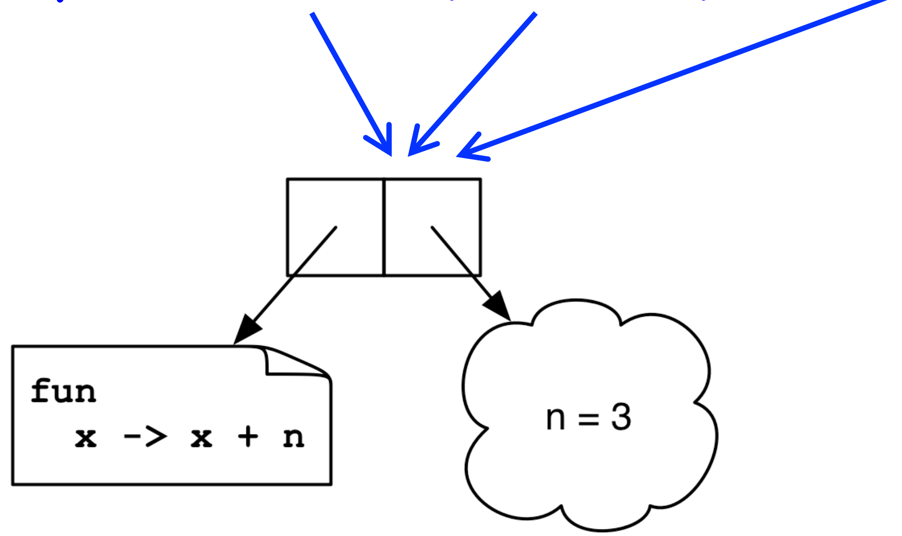
`(mult_sum (3, 4)) 5` → `<cl> 5` → `5 * 7` → `35`



Example – Closure 3

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

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

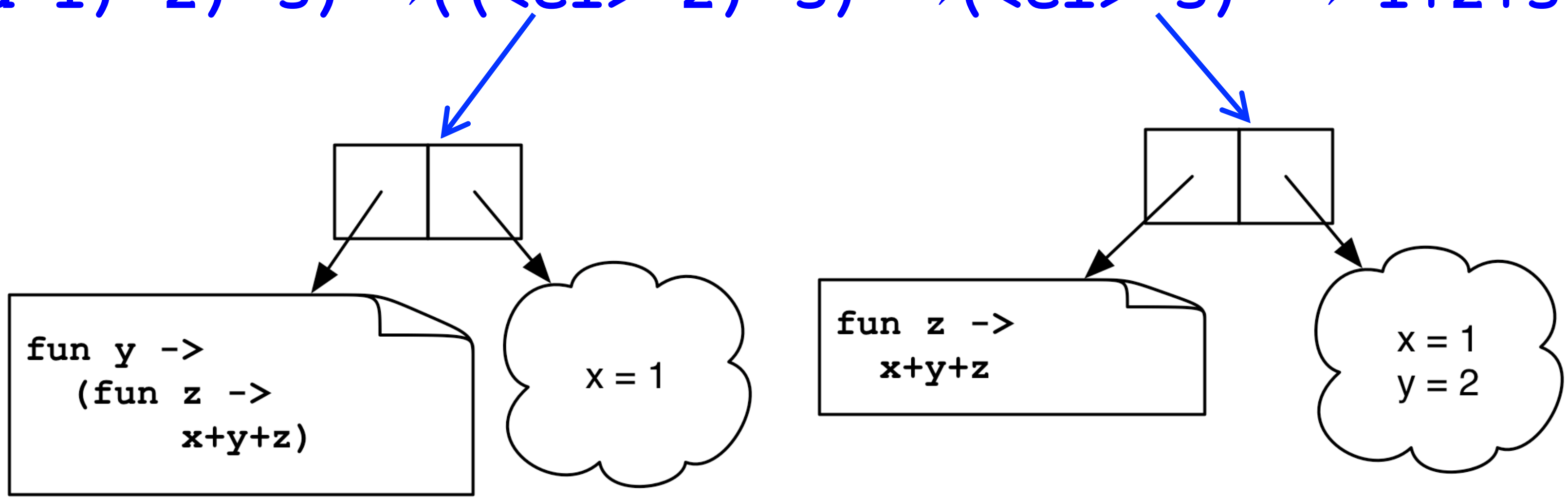


Example – Closure 4

```
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
```



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)
```

Curried Functions in OCaml (cont.)

- What is the type of add?

```
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
  - **`->`** associates to the right
    - **`int -> int -> int`** is the same as **`int -> (int -> int)`**
  - Function application **`( )`** associates to the left
    - **`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
```

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 Functions map & fold

- Map

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

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

- Fold

```
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

```
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] (* [-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 fold Function

- New Fold

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

- Examples

```
let add x y = x + y
fold add 0 [1; 2; 3]     (* 6 *)
let sum = fold add 0
sum [1; 2; 3]            (* 6 *)
let next n _ = n + 1
let len = fold next 0    (* len not polymorphic! *)
len [4; 5; 6; 7; 8]      (* 5 *)
```

- What is the type of this form of fold?

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

Another Convention (cont.)

- 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 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

OCaml 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
```

OCaml Exceptions (cont.)

- 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 reaches the top level

OCaml Exceptions (cont.)

- Exceptions may be thrown by I/O statements
  - Common way to detect end of file
  - Need to decide how to handle exception
- Example

```
try
  (input_char stdin)    (* reads 1 char *)
with End_of_file -> 0   (* return 0?    *)

try
  read_line ()          (* reads 1 line *)
with End_of_file -> “” (* return “”?   *)
```

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 together associated
    - Types, functions, and data
  - Avoid polluting the top-level with unnecessary stuff
- For lots of sample modules
  - See the OCaml standard library

Modularity and Abstraction

- Another reason for creating a module
  - So we can hide details
  - Example
    - Build a binary tree module
    - Hide 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)

Modularity

- Definition
  - Extent to which a computer program is composed of separate parts
  - Higher degree of modularity is better
- Modular programming
  - Programming techniques that increase modularity
    - Interface vs. implementation
- Modular programming languages
  - Explicit support for modules
  - Ada, Fortran, ML, Modula-2, Python, Ruby, OCaml

Creating a Module in OCaml

```
module Shapes =
  struct
    type shape =
       Rect of float * float (* wid*len *)
     | 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;;
```

Creating a Module in OCaml (cont.)

```
module Shapes =
  struct
    type shape = …
    let area = …
    let unit_circle = …
  end;;
unit_circle;;    (* not defined *)
Shapes.unit_circle;;
Shapes.area (Shapes.Rect (3.0, 4.0));;
open Shapes;; (* import names
                 into curr scope *)
unit_circle;;    (* now defined *)
```

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.)

- Convention
  - Signature names are all capital letters
  - 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 – OCaml Module Signatures

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 *)
```

Modules in Java

- Java classes are like modules
  - Provides implementations for a group of functions
  - But classes can also
    - Instantiate objects
    - Inherit attributes from other classes
- Java interfaces are like module signatures
  - Defines a group of functions that may be used
  - Implementation is hidden

Modules in C

- .c files are like modules
  - Provides implementations for a group of functions
- .h files are like module signatures
  - Defines a group of functions that may be used
  - Implementation is hidden
- Usage is not enforced by C language
  - Can put C code in .h file

Module in Ruby

- Ruby explicitly supports modules
  - Modules defined by module … end
  - Modules cannot
    - Instantiate objects
    - Derive subclasses

```
puts Math.sqrt(4)     # 2
puts Math::PI         # 3.1416

include Math          # open Math
puts Sqrt(4)          # 2
puts PI               # 3.1416
```

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
    - Example: Calling a function f with argument x always produces the same result
  - But could take (much) more memory & time to execute

Imperative OCaml

- There are three basic operations on memory

  1) `ref : 'a -> 'a ref`
     - Allocate an updatable reference
  2) `! : 'a ref -> 'a`
     - Read the value stored in reference
  3) `:= : '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 in C (cont.)

Store 3 in location x

Makes no sense

```
int x;
int y;

x = 3;

y = x;

3 = x;
```

Read contents of x and store in location y

- Notice that x, y, 3 all have the same type: **`int`**

Comparison to OCaml

C

```c
int x;
Int y;

x = 3;

y = x;

3 = x;
```

OCaml

```ocaml
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
- But…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 type of variable

- Static or dynamic types?
  - Static – but without type declarations
  - OCaml does type inference to figure out types for you
    - Advantage – less work to write programs
    - Disadvantages – easier to make mistakes, harder to find errors

Higher-Order Functions in C

- C supports function pointers

```
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

```
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 C (cont.)

- Cannot access non-local variables in C
- OCaml code

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

- Equivalent code in C is illegal

```
int (* add(int x))(int) {
  return add_y;
}
int add_y(int y) {
  return x + y; // x undefined
}
```

Higher-Order Functions in C (cont.)

- OCaml code

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

- Works if C supports nested functions
  - Not in ISO C, but in gcc

```
int (* add(int x))(int) {
  int add_y(int y) {
    return x + y;
  }
  return add_y;
}
```

Higher-Order Functions in Ruby

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

```
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

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

```
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

```
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

```
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
```

Higher-Order Functions in Ruby (cont.)

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

```
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