CMSC 330: Organization of Programming Languages

Functional Programming with OCaml

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, 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 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
  
  ```ocaml
  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
- No type declarations
- Need to use correct print function (OCaml also has print)
- 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:
  ```
  ocam1.ml:
  (* A small OCaml program *)
  let x = 37;
  let y = x + 5;
  print_int y;
  print_string "\n";
  ```

  % ocamloccam1.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 = 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 = ()
  ```

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

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
  ```

  - OCaml has static types to help you avoid errors
    - Note: Sometimes the messages are a bit confusing
      # 1 + 5 Emma;
      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

Run, OCaml, Run (cont’d)

- Files can be loaded at the top-level
  ```
  ocam1.ml:
  (* A small OCaml program *)
  let x = 37;
  let y = x + 5;
  print_int y;
  print_string "\n";
  ```

  % ocamloccam1.ml
  % 'use "ocaml1.ml"';
  val x = 37
  val y = 42
  42 : int
  - : unit = ()
  # x ;
  - : int = 37

  %

Run, OCaml, Run (cont’d)

- fuse loads in a file one line at a time
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`

  ```ocaml
  let pi = 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

  ```ocaml
  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 *)

  ```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 */

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

  ```ocaml
  list parameters after function name
  no parentheses on function calls
  no return statement
  ```

Defining Functions

use `let` to define functions

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

  ```ocaml
  def fn (x) = (float_of_int x) * .3.14
  ```

  ```ocaml
  print_string (* type string -> unit *)
  ```

  no return statement

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`

  ```ocaml
  let next x = x + 1 (* type int -> int *)
  let fn x = (float_of_int x) *.3.14
  ```

  ```ocaml
  print_string (* type string -> unit *)
  ```

  no return statement

Local Variables

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

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

  – And you can use as many `lets` as you want

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

- Very useful for debugging, especially for more complicated types

Lists in OCaml

- The basic data structure in OCaml is the list
  - Lists are written as `[e1; e2; ...; en]`
  - 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

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

Lists in OCaml are Linked

```
[1; 2; 3]  
```
  - [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)

```
1 2 3  
```

- `::` 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::  
  2::(3::)  
  1::(2::(3::))  
  ```
More Examples

```ocaml
# 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]
# let w = [1;2]::y;;
This expression has type int list but is here used with type int list list
• not modifying existing lists, just creating new lists
```

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
  ```ocaml
  match e with pl -> el | ... | pn -> en
  ```
  - `pl...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

```ocaml
match e with pl -> el | ... | 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;;
```

```
# 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 *)

### An Abbreviation

- 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

- Useful if there's only one acceptable input

### Pattern Matching Lists of Lists

- You can do pattern matching on these 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
  - We'll see that soon

### OCaml Functions Take One Argument

- Recall this example
  ```ocaml
  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 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) = (x1, y1, z1)
    - plusThree (addOne (3, 4, 5); (* returns 15 *)

- let sum ([x, b], c) = (x+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 t1s (_::xs, _::ys) = (xs, ys)
  - t1s [[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

### Another Example

- let f l = match l with x::_::y -> (x,y)

- What is f [1;2;3;4]?

- Possibilities: 
  - (11,[3])
  - (1,3)
  - (1,[3])
  - (1,4)
  - (1,[3;4])
**List and 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)] :`

**Type declarations**

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

**Examples**

```ocaml
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
```

**Examples of Polymorphic Types**

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

- But other functions work for any list
  - `let hd (h::_::s) = h`
  - `hd [1; 2; 3] (* returns 1 *)`
  - `hd ["a"; "b"; "c"] (* returns "a" *)`

- OCaml gives such functions polymorphic types
  - `hd : 'a list -> 'a`
  - this says the function takes a list of any element type `'a`, and returns something of that type

**Tuples Are a Fixed Size**

```ocaml```
```
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
## Conditionals

- Use `if...then...else` just like C/Java
  - No parentheses and 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 7 > 42 then "hello" else "goodbye";;
    # let x = if true then 3 else 4;;
    # 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 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;;` (* 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` *)
- `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

  ```ocaml
  let rec print_up_to (n, m) =
    if n < m then print_up_to (n + 1, m)
  let rec print_up_to (n, m) =
    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
    | (...) -> 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 els 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)
  ```

- last l (* last element of l *)
  
  ```ocaml
  let rec last l = match l with
  [x] -> x
  | (x::xs) -> last xs
  ```

More Examples (cont’d)

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

- rev l (* reverse list; hint: use append *)
  
  ```ocaml
  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, (a::x))
  - let rev l = rev_helper (l, [])

  * Let’s give it a try
    
    ```ocaml
    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 *)
  
  ```ocaml
  let rec flattenPairs l = match l with
  [] -> []
  | ((a, b)::t) -> a :: b :: (flattenPairs t)
  ```

- take (n, l) (* return first n els of l *)
  
  ```ocaml
  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:

    ```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

```ocaml
let plus_three x = x + 3
let twice (f, x) = f (f x)
twice (plus_three, 5)
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
pick_fn : int -> (int->int)
```

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

```ocaml
map ((fun x -> x + 13), [1; 2; 3])
twice ((fun x -> x + 2), 4)
```

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

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

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

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

Pattern Matching with fun

• `match` can be used within `fun`

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

  – For complicated matches, though, use named functions

• Standard pattern matching abbreviation can be used

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

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 x ->
      (match x 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

```ocaml
defold (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

```ocaml
defadd (a, x) = a + x
defold (add, 0, [1; 2; 3; 4])
defold (add, 1, [2; 3; 4])
defold (add, 3, [3; 4])
defold (add, 6, [4])
defold (add, 10, [])
defold (add, 10, [])
```

We just built the sum function!

Another Example

```ocaml
defnext (a, _) = a + 1
defold (next, 0, [2; 3; 4; 5])
defold (next, 1, [3; 4; 5])
defold (next, 2, [4; 5])
defold (next, 3, [5])
defold (next, 4, [])
```

We just built the length function!

Using fold to Build rev

```ocaml
defprepend (a, x) = x : a
defold (prepend, [], [1; 2; 3; 4])
defold (prepend, [1], [2; 3; 4])
defold (prepend, [2; 1], [3; 4])
defold (prepend, [3; 2; 1], [4])
defold (prepend, [4; 3; 2; 1], [])
defold (prepend, [4; 3; 2; 1], [])
```

We just built the reverse function with fold?

The Call Stack in C/Java/etc.

```c
void f(void) {
    int a;
    a = g(3);
}
int g(int x) {
    int y;
    y = h(x);
    return y;
}
int h(int x) {
    return x + 1;
}
int main() {
    f();
    return 0;
}
```

Nested Functions

- In OCaml, you can define functions anywhere
  - Even inside of other functions

```ocaml
defold (fun [a, x] => a + x), 0, 1)
defold (fun [a, x] => a + x), 0, 1)
defold (fun [a, x] => a + x), 0, 1)
```

```ocaml
let pick_one n =
    if n > 0 then (fun x => x + 1)
    else (fun x => x - 1)
    (pick_one n) 6 (* returns 5 * )
```
Nested Functions (cont’d)

- You can also use `let` to define functions inside of other functions

```ocaml
let sum l =
  let add (x, 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
```

Consider the Call Stack Again

- Uh...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

```ocaml
let map [f, n] = match n with
  [] -> []
  [h; t] -> [h; f (map (f, t))]
let addN (n, l) =
  let add x = n + x in
  map (add, l)
addN [3; [1; 2; 3]]
```

Returned Functions

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

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

How About This?

- (Equivalent to...)

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

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:

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

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**

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

(\[\text{add } 3 \text{ 4 } \Rightarrow \text{<closure> 4 } \Rightarrow \text{ 3 + 4 } \Rightarrow \text{ 7}\])

**Another Example**

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

(\[\text{mult\_sum } (3, 4) \Rightarrow \text{<closure> 5 } \Rightarrow \text{ 5 * 7 } \Rightarrow \text{ 35}\])

**Yet Another Example**

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

(\[\text{twice } (3, 4) \Rightarrow \text{<closure> (<closure> 4) } \Rightarrow \text{<closure> 7 } \Rightarrow \text{ 10}\])

**Still Another Example**

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

(((\text{add } 1) \text{ 2}) \text{ 3}) \Rightarrow (\text{<closure> 2}) \text{ 3} \Rightarrow (\text{<closure> 3}) \Rightarrow \text{ 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
  ```ocaml
  let add x y = x + y
  ```
  - This is identical to all of the following:
    ```ocaml
    let add x = (fun y -> x + y)
    let add = (fun x y -> x + y)
    let add = (fun x y z -> x + y)
    ```

- Thus:
  ```ocaml
  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) -> f (f a) h
```

```
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/ocaml/lib/ocaml` on Grace
    - In particular, look at the file `list.ml` for standard list functions
    - Access these functions using `List.<fn name>`
    - E.g., `List.hd`, `List.length`, `List.map`

- OCaml plays a lot of tricks to avoid creating closures and to avoid allocating on the heap
  - It's unnecessary much of the time, since functions are usually called with all arguments

Higher-Order Functions in C

- C has function pointers but no closures
  - (gcc had closures)

```c
typedef int (*int_func)(int);
void app(int_func f, int *a, int n) {
    for (i = 0; i < n; i++)
        a[i] = f(a[i]);
}
int add_one(int x) { return x + 1; }
int main() {
    int a[4] = {1, 2, 3, 4};
    app(add_one, a, 4);
}
```

Higher-Order Functions in Ruby

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

```ruby
def my_collect(a)
    b = Array.new(a.length)
    i = 0
    while i < a.length
        b[i] = yield(a[i])
        i += 1
    end
    return b
end
b = my_collect([1, 2, 3, 4, 5]) { |x| -x }
```

Higher-Order Functions in Ruby (cont.)

- Ruby code blocks are actual variables

```ruby
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
end
quad { x += 1 } # same as x += 4
```

Higher-Order Functions in Ruby (cont.)

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

```ruby
c1 = Proc.new { x+1 }
c2 = proc { x+1 }
c3 = lambda { x+1 }
def foo
    x+=1
end

c4 = method { :foo }
    1
    c.call # x+=1
```
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

Data Types

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

let area x =
  match x with
  | Rect (w, l) -> w *. l
  | Circle r -> r *. r *. 3.14
  area (Rect (3.0, 4.0))
```

- `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)

```ocaml
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

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 (cont’d)

```ocaml
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?

Polymorphic Data Types

```ocaml
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?

```ocaml
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

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

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

### Exercise: A Binary Tree Data Type

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

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

### Exceptions

- 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

```ocaml
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 e ->
    Printf.printf "Caught %s\n" e
```

### 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

```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 * 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 *)
  Shapes.area (Shapes.Rect (3.0, 4.0));
```

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

```ocaml
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 *)
```

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

```ocaml
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 \textit{shape} is hidden

Abstract Types in Signatures (cont’d)

```ocaml
# 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 = 28.68

# Shapes.make_circle 3.0

# (* 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

```ocaml
module type shape = 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 *)
```

shapes.ml

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

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

```ocaml
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

```ocaml
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
- 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)

• Notice that x, y, and 3 all have type int

\[
\begin{align*}
\text{int } x, y; \\
x &= 3; \\
y &= x; \\
3 &= x;
\end{align*}
\]

Capturing a ref in a Closure

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

\[
\begin{align*}
\text{let next =} \\
\text{let count = ref 0 in} \\
\text{function () =>} \\
\text{let temp = !count in} \\
\text{count := (!count) + 1; \\
\text{temp; \\
}\#\text{ next () ;} \\
\text{=} \text{ : int } = 0 \\
\text{# next ();} \\
\text{=} \text{ : int } = 1
\end{align*}
\]

Comparison to OCaml

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

\[
\begin{align*}
\text{int } x, y; \\
\text{let x = ref 0;;} \\
x &= 3;; \\
\text{(* x : int ref *)} \\
\text{y := (\text{!x});} \\
3 &= x;; \\
\text{(* 3 : int; error *)}
\end{align*}
\]

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

\[
\begin{align*}
\text{let x = ref 0} \\
\text{let f () =} \\
\text{begin} \\
\text{print_string “hello”; \\
x := (!x) + 1 \\
\text{end}
\end{align*}
\]

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

```ocaml
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

```ocaml
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!

```ocaml
# let x = 42 ;;
val x : int = 42
# x = x + 1 ;;
val x : int = 42
       (* 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!

```ocaml
# let x = 42 ;;
val x : int = 42
# let f y = y + x ;;
val f : int -> int = <fun1>
# let x = 0 ;;
val x : int = 0
       (* shadow binding of x *)
# x 0 ;
val x : int = 0
       (* but f still refers to x=42 *)
```