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 printf)

;; ends a top-level expression

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

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

- Compiling and running the previous small program:

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

```bash
% ocamlc ocaml1.ml
% ./a.out
42
%
```
Expresssions 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 = ()
```

"-" = “the expression you just typed"

unit = “no interesting value” (like void)

gives type and value of each expr
Run, OCaml, Run (cont.)

- Files can be loaded at the top-level

```ocaml
% ocaml

Objective Caml version 4.00.1

# #use "ocaml1.ml";;
val x : int = 37

val y : int = 42

42- : unit = ()

- : unit = ()

# x;;
- : int = 37
```

ocaml1.ml:

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

#use loads in a file one line at a time
A Note on `;;`

- `;;` 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

- There is also a single semi-colon `;` in OCaml
  - But we won’t need it for now
  - It’s only useful when programming imperatively, i.e., with side effects
    - Which we won’t do for a while
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

  ```ocaml
  # 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
Defining Functions

- Use `let` to define functions.
- List parameters after the function name.
- No parentheses on function calls.
- No return statement.

```ml
let next x = x + 1;;
next 3;;
let plus (x, y) = x + y;;
plus (3, 4);;
```
Local Let Bindings

• 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
```

– Notice the use of in --- this is a local let
Semantics Of Local Let

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

\[
let \ pi = 3.14 \ in \ pi \ .* \ 3.0 \ .* \ 3.0;;
pi;;
\]

bind pi in body of let
floating point multiplication
error
More On Local Lets

• Compare to similar usage in Java/C

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

```java
{  
  float pi = 3.14;
  
  pi * 3.0 * 3.0;

}  
pi;
```

• In the top-level, omitting `in` means “from now on”:

```ocaml
# let pi = 3.14;;
```

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

• 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 *)

{
  float pi = 3.14;
  float r = 3.0;

  pi * r * r;
}
/* pi, r not in scope */
```
Examples – Let (Local and Toplevel)

• `x;;`
  – (* Unbound value x *)

• `let x = 1 in x + 1;;`
  – (* 2 *)

• `let x = x in x + 1;;`
  – (* Unbound value x *)
Examples – Let (Local and Toplevel)

• let x = 1 in (x + 1 + x) ;;
  – (* 3 *)

• (let x = 1 in x + 1) ;; x;;
  – (* Unbound value x *)

• let x = 4 in (let x = x + 1 in x);
  – (* 5 *)
Function Types

• In OCaml, \( \rightarrow \) is the function type constructor
  – The type \( t_1 \rightarrow t_2 \) is a function with argument or \textit{domain} type \( t_1 \) and return or \textit{range} type \( t_2 \)

• Examples
  – let next x = x + 1 (* type int \rightarrow int *)
  – let fn x = (float_of_int x) *. 3.14 (* type int \rightarrow float *)
  – print_string (* type string \rightarrow unit *)

• Type a function name at top level to get its type
Type Annotations

• The syntax \((e : t)\) asserts that “\(e\) has type \(t\)”
  – This can be added anywhere you like
    \[
    \text{let } (x : \text{int}) = 3 \\
    \text{let } z = (x : \text{int}) + 5
    \]

• Use to give functions parameter and return types
  \[
  \text{let fn (x:int):float =} \\
  (\text{float_of_int x}) \ast . 3.14
  \]
  – Note special position for return type
  – Thus \(\text{let } g x:\text{int} = \ldots\) means \(g\) returns \(\text{int}\)

• 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 \([e_1; e_2; \ldots; e_n]\)
    
    # [1;2;3]
    
    - : int list = [1;2;3]
  – Notice \texttt{int list} – lists must be \textit{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;
...
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 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

```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]

• *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?
```
Digression: Shadowing

- If you bind the same variable twice, the most recent is in play
  - Looks like variable assignment, but it is not

- let x = [1; 2];;
- let y = 3::x;;
- let x = [3];; (* shadows x *)
- y;;
  - (* [3; 1; 2] *)
- x;;
  - (* [3] *)
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] \[\text{int list}\]
  – [ [ [ ]; [ ]; [1.3;2.4] ] ] \[\text{float list list list}\]
  – let func x = x::(0::[ ]) \[\text{int -> int list}\]
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 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
  – Does not add any bindings
  – Useful when you want to know something matches, but don’t care what its value is
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 [ ] *)
Missing Cases

- Exceptions for inputs that don’t match any pattern
  – OCaml will warn you about non-exhaustive matches

Example:

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

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

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

• let g l =
  match l with [h1; h2] -> h1 + h2
  - g [1; 2]
  - (* evaluates to 3 *)
  - g [1; 2; 3]
  - (* error! no pattern matches *)
Pattern Matching – An Abbreviation

- \texttt{let f p = e}, where \( p \) is a pattern
  - is shorthand for \texttt{let f x = match x with p -> e}

- Examples
  - \texttt{let hd (h::_)} = \( h \)
  - \texttt{let tl (_,:t)} = \( t \)
  - \texttt{let f (x::y::_)} = \( x + y \)
  - \texttt{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; 3]; [4; 5]; [7; 8; 9] ] = 5
  - let addFirstSecond ((x::_)::(_::y::_)::_) = x + y
    - addFirstSecond [ [1; 2; 3]; [4; 5]; [7; 8; 9] ] = 6

- **Note:** You probably won’t do this much or at all
  - You’ll mostly write recursive functions over lists
  - We’ll see that soon
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 \((e_1, \ldots, e_n)\)
• 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
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) = Warning: This expression should have type unit
Another Example

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

- let eq (x,y) = x = y
  - eq : 'a * 'a -> bool
Tuples Are A Fixed Size

- This OCaml definition
  
  ```ocaml
  # 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
Conditionals

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

```python
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

```ocaml
# 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?

```ml
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

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

Fact is not bound here!
Let – More Examples

- let \( f \ n = 10 \);
  - let \( f \ n = \text{if } n = 0 \text{ then } 1 \text{ else } n \times f \ (n - 1) \);
    - \( f \ 0 \); (* 1 *)
    - \( f \ 1 \); (* 10 *)

- let \( f \ x = \ldots \ f \ldots \ \text{in} \ldots \ f \ldots \)
  - (* Unbound value f *)

- let rec \( f \ x = \ldots \ f \ldots \ \text{in} \ldots \ f \ldots \)
  - (* Bound value f *)
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) =
    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 nonempty list is 1 plus the length of the tail

  – Type of `length`?
More Examples

• sum l (* sum of elts in l * )
  let rec sum l = 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)

• last l (* last element of l *)
  let rec last l = match l with
   [x] -> x
   | (x::xs) -> last xs
More Examples (cont.)

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

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

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

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

- rev takes $O(n^2)$ time. Can you do better?
A Clever Version of Reverse

```ocaml
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

  ```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 *)
  let rec flattenPairs l = match l with
      []          -> []
      | ((a, b)::t) -> a :: b :: (flattenPairs t)

• take (n, l) (* return first n elts of l *)
  let rec take (n, l) =
      if n = 0 then []
      else match l with
          []          -> []
          | (x::xs)    -> x :: (take (n-1, xs))
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?