CMSC 330: Organization of Programming Languages

Functional Programming with OCaml
Review

• function declaration
  - let [rec] <name> <inputs> = <output>

• types
  - int, bool, string

• lists
  - [i1; i2; i3] or i1::i2::i3::[]
Pattern Matching

- To pull lists apart, use the `match` construct
  
  ```
  match e with p_1 -> e_1 | ... | p_n -> e_n
  ```

- `p_1...p_n` are *patterns* made up of `[]`, `::`, and *pattern variables*

- `match` finds the first `p_k` that matches the shape of `e`
  - Then `e_k` is evaluated and returned
  - During evaluation of `e_k`, pattern variables in `p_k` 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

\[
\text{match } e \text{ with } p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n
\]

\[
\text{let is_empty } l = \text{match } l \text{ with}
\]
\[
[\] \rightarrow \text{true}
\]
\[
(\text{h::t}) \rightarrow \text{false}
\]

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:
  
  ```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:
  []
  ```
Example

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

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

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

Two element list [h1;h2]
An Abbreviation

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

- **Examples**
  - \texttt{let \( \text{hd} \ (\text{h}::\_\_\_\_.) = \text{h} \)}
  - \texttt{let \( \text{tl} \ (\_\_\_\text{::t}) = \text{t} \)}
  - \texttt{let \( f \ (\text{x}::\text{y}::\_\_\_.) = \text{x} + \text{y} \)}
  - \texttt{let \( g \ [\text{x}; \ \text{y}] = \text{x} + \text{y} \)}

- 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

```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 \((e_1, \ldots, e_n)\)
  – They’re like C structs but without field labels, and allocated on the heap
  – Unlike lists, tuples do not need to be homogenous
  – E.g., \((1, ["string1"; "string2"]]\) is a valid tuple

• Tuples are deconstructed using pattern matching
Examples with Tuples

• let plusThree \((x, y, z) = x + y + z\)
  let addOne \((x, y, z) = (x+1, y+1, z+1)\)
  – plusThree (addOne (3, 4, 5)) (* returns 15 *)

• let sum \(((a, b), c) = (a+c, b+c)\)
  – sum ((1, 2), 3) = (4, 5)

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

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

• Remember, semicolon for lists, comma for tuples
  – [1, 2] = [[1, 2]] = a list of size one
  – (1; 2) = a syntax error
Another Example

• let f l = match l with x::(_:y) -> (x,y)
• What is f [1;2;3;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') :
  - [(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
Type declarations

• **type** can be used to create new names for types
  – useful for combinations of lists and tuples
  – like “typedef” in C

• Examples
  
  ```
  type my_type = int * (int list)
  (3, [1; 2]) : my_type
  
  type my_type2 = int * char * (int * float)
  (3, ‘a’, (5, 3.0)) : my_type2
  ```
Polymorphic Types

• 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
Tuples Are a Fixed Size

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

• Thus there's never more than one match case
  with tuples
    – How’s this instead?
    let foo (a, b) = a + b
Conditionals

• Use `if...then...else` like C/Java
  – No parentheses and no end

```plaintext
if grade >= 90 then
  print_string "You got an A"
else if grade >= 80 then
  print_string "You got a B"
else if grade >= 70 then
  print_string "You got a C"
else
  print_string "You’re not doing so well"
```
Conditionals (cont’d)

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

    # if 7 > 42 then "hello" else "goodbye";;
    - : string = "goodbye"

    # let x = if true then 3 else 4;;
    x : int = 3

    # if false then 3 else 3.0;;
    This expression has type float but is here used
    with type int

• Putting this together with what we’ve seen earlier, can you write fact, the factorial function?
The Factorial Function

```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`
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

```ml
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 function?
More examples of let (try to evaluate)

- let x = 1 in x;;
- let x = x in x;;
- let x = 4 in
  let x = x + 1 in x;;
- let f n = 10;;
  let f n = if n = 0 then 1 else n * f (n - 1);;
    f 0;;
    f 1;;
- let g x = g x;;
More examples of let

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