OCaml Features

Pattern Matching Review

Match syntax

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

Code 1

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

Code 2

```
  let hd l = match l with (h::t) -> h
```

Code 3

```
  let tl l = match l with (h::t) -> t
```

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:::_::t) -> 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

```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).
```

Pattern Matching – An Abbreviation

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

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

```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

```ocaml
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 heterogeneous
  - Unlike lists, which must be homogenous
  - \((1, \text{["string1"; "string2"]})\) is a valid tuple

Tuples – Examples

- let plusThree \((x, y, z) = x+y+z\)
  - let addOne \((x, y, z) = (x+1, y+1, z+1)\)
  - \(\text{plusThree (addOne (3,4,5))} = 15\)
- let sum \(((a, b), c) = (a+c, b+c)\)
  - \(\text{sum ((1, 2), 3) = (4,5)}\)
- let plusFirstTwo \((x::y::_, a) = (x+a, y+a)\)
  - \(\text{plusFirstTwo ([1; 2; 3], 4) = (5,6)}\)

Tuples – More Examples

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

Remember

- Semicolon for lists
- Comma for tuples

Example

- \([1, 2] = [1, 2]\) is a list of size one
- \((1, 2)\) is 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

Type declarations

 type can be used to create new names for types

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

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

  - Thus never more than one match case with tuples

Conditionals

- Use `if...then...else` just like C/Java
  
  ```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.)

- 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`
  - `let rec x = e1 in e2`

  Why use `let rec`?
  - If you used `let` instead of `let rec` to define `fact`
    ```ml
    let fact n =
      if n = 0 then 1
      else n * fact (n-1) in e2
    ```
    Fact is not bound here!

Examples – Semicolon

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

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 x;;
  - (* Unbound value f *)