Lecture 34:

- OCaml: Pattern matching

Announcements:

- HW-7 due Friday

OCaml Pattern Matching

Basic form of a match expression ... definition by cases

```
match e with
| p1 -> e1
| p2 -> e2
| ...
| pn -> en
```

- e is expression to match, p_i are value "patterns", e_i are result expressions
- ... the first | is optional
- e matched with p_1, then p_2, etc., until a match is found
- first match with p_i results in e_i (and matching stopped)

Simple example where there are just two possible value matches:

```
let my_not x =
  match x with
  | true -> false
  | false -> true
```

OCaml Pattern Matching

Another simple example ...

```
let vowel x =
  match x with
  | 'a' -> true
  | 'e' -> true
  | 'i' -> true
  | 'o' -> true
  | 'u' -> true
  | _ -> false

(* simplifying the pattern *)
let vowel x =
  match x with
  | 'a' | 'e' | 'i' | 'o' | 'u' -> true
  | _ -> false
```

- the _ (underscore) matches any value (aka a "don't care")

OCaml Pattern Matching

Patterns should be "exhaustive" ...

```
let vowel x =
  match x with
  | 'a' -> true
  | 'e' -> true
  | 'i' -> true
  | 'o' -> true
  | 'u' -> true

Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
'b'
val vowel : char -> bool = <fun>

# vowel 'b';;
Exception: Match_failure ("//toplevel//", 2, 0).
```

OCaml Pattern Matching

Pattern Type Constraints

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

- Each p_i must have the same type as e
- Each e_i must have the same type

Pattern matching with lists

... const (`::`) "deconstructs"

- `[]` matches the empty list
- `[x]` matches a one-element list ... also `x::[]`
- `[x; y]` matches a two-element list, etc. ... also `x::y::[]`
- `h::t` matches a list with at least one element (head, tail)
- `h1::h2::t` matches a list with at least two elements, etc.

OCaml Pattern Matching

Examples:

```
let empty xs =
  match xs with
  | [] -> true
  | _ -> false

let rec length xs =
  match xs with
  | [] -> 0
  | _::t -> 1 + length t

let rec shuffle xs ys =
  match (xs, ys) with            (* construct pair to match two values *)
  | ([], ys') -> ys'
  | (xs', []) -> xs'
  | (x'::xs', y'::ys') -> x' :: y' :: mix xs' ys'
```

Q: Are the patterns for `shuffle` "exhaustive"? What about `[] []`?

OCaml Pattern Matching

Note that `match` is just an expression ...

```
# match [1; 2] with
  | [] -> 0
  | [h] -> 1
  | h1::h2::[] -> 2 (* also two elem list *)
  | _ -> 3 ;;
```

Q: What is wrong with this match expression?

```
# match [1; 2; 3] with
  | [] -> 0
  | h::t -> 1
  | h1::h2::t -> 2 ;;
```

- the third pattern will never match ... order of patterns is important!

OCaml will warn you ...

```
Warning 11 [redundant-case]: this match case is unused.
- : int = 1
```


OCaml Pattern Guards

Guards allow us to define conditions for a pattern

| p_i when c_i -> e_i

Example: Get the i-th element in a list

```
let rec get i xs =
  if i < 0 || i >= List.length xs then failwith "Invalid Index"
  else if i == 0 then List.hd xs
  else get (i-1) (List.tl xs)

let rec get i xs =
  match xs with
  | [] -> failwith "Invalid Index"
  | x::t -> if i == 0 then x else get (i-1) (tail xs)

 let rec get i xs =
   match xs with
   | [] -> failwith "Invalid Index"
   | x::_ when i == 0 -> x
   | _::t -> get (i-1) t
```

OCaml Pattern Guards

Example with multiple guards ...

```
let grade p =
  match p with
   | _ when p >= 0.9 -> "A"
   | _ when p >= 0.8 -> "B"
   | _ when p >= 0.7 -> "C"
   | _ when p >= 0.6 -> "D"
   | _ -> "F"
```

Can still have non-exhaustive pattern warnings (and errors):

```
# let grade p =
    match p with
     | _ when p >= 0.9 -> "A"
     | _ when p >= 0.8 -> "B"
     | _ when p >= 0.7 -> "C"
     | _ when p >= 0.6 -> "D"
  ;;
  Warning 8 [partial-match]: this pattern-matching is not exhaustive.
  All clauses in this pattern-matching are guarded.
  val grade : float -> string = <fun>
```