CMSC 330: Organization of Programming Languages

Functional Programming with OCaml, con't.

Examples with Tuples

- `let plus_three (x, y, z) = x + y + z`
  `let add_one (x, y, z) = (x+1, y+1, z+1)`
  - `plus_three (add_one (3, 4, 5))    (* returns 15 *)`
- `let sum ((a, b), c) = (a+c, b+c)`
  - `sum ((1, 2), 3) = (4, 5)`
- `let plus_first_two (x::y::_, a) = (x + a, y + a)`
  - `plus_first_two ([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's `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`

Tuples Are a Fixed Size

```
# let f 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

Type declarations

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

- 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 plus_first_two (x::y::_, a) = (x + a, y + a)`
  - `plus_first_two : 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`

Conditionals

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

```
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, Ruby, 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

```
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 fn n = 10;;
  let fn n = if n = 0 then 1 else n * fn (n - 1);;
  fn 0;;  (* 1 *)
  fn 1;;  (* 10 *)
• let fn x = fn x;;      (* error since fn is not
                            already defined *)
```

Recursion = Looping

- Recursion is essentially the only way to iterate
  - (the only way we're going to talk about)
- Another example

```
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

```
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 its tail*
  - type of length?

More Examples

```
sum l (* return the sum of the elements in list l *)
   let rec sum l = match l with
       [] -> 0
     | (h::t) -> h + (sum t)

negate l (* return new list with negated elements of l *)
    let rec negate l = match l with
       [] -> []
     | (h::t) -> (-h) :: (negate t)

last l  (* return last element of nonempty list l *)
   let rec last l = match l with
       [h] -> h
     | (h::t) -> last t
```


More Examples (cont'd)

```
(* 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
     | (h::t) -> h::(append (t, m))

rev l (* return reverse of list l; hint: use append *)
   let rec rev l = match l with
       [] -> []
     | (h::t) -> append ((rev t), [h])
```

- **rev** takes $O(n^2)$ time. Can you do better?

A More Clever Version of Reverse

```
let rec rev_helper (l, a) = match l with
    [] -> a
  | (h::t) -> rev_helper (t, (h::a))
let rev l = rev_helper (l, [])
```

- Let's give it a try

```
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]
```