Lecture 2 — OCaml basics and recursive functions on lists

- Ocaml basics
  - Let expressions and scope
  - Functions on tuples
  - Pattern-matching
- Recursive functions on lists: the recursion fairy

Let expressions

- The let expression is fundamental in OCaml because it is how variables are declared.
- It is used both at the top level and within other expressions.
- Top-level forms (introduced in last class):
  - `let x = expr;;`
  - `let f args = expr;;`
  - `let rec f args = expr;;`

Let expressions (cont.)

- **Non-top-level (nested) let expressions:**
  - `let x = expr in expr`
  - `let f args = expr in expr`
  - `let rec f args = expr in expr`
- **Examples:**
  - `let qmark = "?" in "Wha"^qmark`
  - `let abs x = if x<0 then -x else x`
    `in abs (getnum file)`
  - `(let x=1 in x+3) + (let y=2 in 2*y)`

Let expressions (cont.)

- **Examples:**

```
let sumsqrs x y = let sqr a = a*a
                  in sqr x + sqr y;;

let binom n m = let rec fac x = if x=0 then 1 else x * fac (x-1)
                in fac n / (fac m * fac (n-m));;
```

- **Let expressions introduce names for values; The names are called *variables*, or sometimes, *let-bound variables*. But they are not really variables in the sense of imperative languages: there is no "assignment" to these names. If another let expression uses the same name, they are distinct variables (like when you have a local variable in Java that has the same name as a field).**

Let expression exercises

Give the value of each expression:

```
let x = 3 in x+x

let x = 3 in let y = 4 in x+y

let x = 3 in let dbl y = y*2 in dbl x

let x = let f a = a+1 in f (f 3) in x

let f a b = let y = a*b in y*a in f 3 4


let g x = let rec wha a = if a=0 then 1 else 1 + wha(a-1)
          in wha (wha x)
in g 7
```

Scope in OCaml

"Scope" means: in what region of the program can a particular name be used?

- **Top-level let expressions (assume all in a file; in the following, P refers to the rest of the file after the given let expression):**
  - `let` x `=` e`;;` P — **scope of x is P**
  - `let` f x `=` e`;;` P — **scope of x is e; scope of f is P**
  - `let rec` f x `=` e`;;` P

    — **scope of x is e; scope of f is e and P**

Scope in OCaml (cont.)

- **Nested let expressions:**
  - `let` x `=` e `in` e' — **scope of** x **is** e'
  - `let` f x `=` e `in` e' — **scope of** x **is** e**; scope of** f **is** e'
  - `let rec` f x `=` e `in` e'

    — **scope of** x **is** e**; scope of** f **is** e **and** e'

- **Note: if we just put "`in`" in front of all top-level definitions (except the first), and remove the semicolons, the program becomes one large `let` expression; the scope rules would be the same.**

Scope exercises

Consider the let expressions from the last exercise, with some variable names changed. Give the value of each expression:

```
let x = 3 in x+x

(let x=1 in x+3) + (let x=2 in 2*x)

let x = 3 in let x = 4 in x+x

let x = 3 in let dbl x = x*2 in dbl x

let x = let f x = x+1 in f (f 3) in x

let f a b = let b = a*b in b*a in f 3 4

let g x = let rec wha x = if x=0 then 1 else 1 + wha(x-1)
          in wha (wha x)
in g 7
```

Mutual recursion

What do these top-level definitions do:

```
let rec even n = if n=0 then true else odd(n-1);;
let rec odd n = if n=0 then false else even(n-1);;
```

Pattern-matching

- When variables are introduced — in let expressions — can instead use a *pattern* that describes the form of the value, and allows more than one variable to be matched.

- For example, here are three ways to write the identical function, which adds the two members of an int * int pair:

  - `let sum p = fst p + snd p`
  - `let sum (a,b) = a+b`
  - `let sum p = let (a,b) = p in a+b`

Pattern-matching (cont.)

- **Pattern-matching allows us to define functions on larger tuples:**
  - **Ex:** `fst_of_3` **returns the first member of a triple, e.g.** `fst_of_3 (4.0, 3, 2) = 4.0`**. Define it in two different ways:**

Curried vs. uncurried functions

Consider two similar function definitions:

```
let sum1 x y = x+y;;
let sum2 (x,y) = x+y;;
```

- **Show a correct call to each of these functions:**

- **Give the type of each function:**

- **What happens if you enter `sum1(3,4)` or `sum2 3 4`?**

- **`sum1` is in "curried" form, `sum2` in "uncurried" form. Either form can be used, but curried form is more common in OCaml.**

"match" expressions

- **match expressions are used to match a pattern to a value. They give another way to define `sum`:**

```
let sum p = match p with
              (a,b) -> a+b;;
```

- **Match expressions are powerful because they allow a function to be defined with a sequence of alternatives, which give a more elegant syntax than conditional expressions.**

```
let rec fac n = match n with
                  0 -> 1                    (* match 0 *)
                | _ -> n * fac(n-1)         (* match anything else *)
```

Functions on lists

- **Pattern-matching is used commonly to define functions on lists.**
- **E.g. define** `hd`**:** `let hd (h::t) = h`
- **E.g. addfirsttwo: int list** $\rightarrow$ **int adds first two elements of a list:** `let addfirsttwo (h::ht::tt) = h+ht`
- **Ex: Define rev2, which switches the first two elements of a list:** `rev2 [2;3;4;5] = [3;2;4;5]`**:**

Functions on lists (cont.)

- **Most often, list functions are defined using match expressions with more than one clause, e.g. one clause for the empty list and one for non-empty lists. Here are two equivalent definitions of a function:**

```
let rec length lis = if lis=[] then 0 else 1 + length (tl lis)

let rec length lis = match lis with
                       [] -> 0
                     | h::t -> 1 + length t
```

- **Aside: when a variable in a pattern isn't used, it can be replaced by an underscore.**

Functions on lists (cont.)

- **Ex: second: int list $\rightarrow$ int returns 0 for an empty list, the head of a one-element list, and the second element of any other list. Define it with and without match expressions:**

```
let second lis = if lis==[] then




let second lis = match lis with

    [] ->
```

The recursion fairy

- **Suppose you want to write a function `f` on lists. This is the easiest way:**
  - ***Assume*** **you are given r = `f (tl x)` (by the recursion fairy!)**
  - **Figure out how you can calculate `f x` from r and `hd x` (and only those two things).**
  - **Then you're almost done: Define `f` as:**

```
let rec f x = match x with
                [] -> fill in base case
              | h::t -> calculate f x from h and f t
```

Ex: sum

- **Define `sum`: int list $\rightarrow$ int that adds up the elements of a list.**
- **First: To calculate `sum lis`, suppose s = the sum of the elements in `tl lis`. What is the sum of all the elements in `lis`?**

- **Second: Define `sum`:**

```
let rec sum lis = match lis with
        [] ->
      | h::t ->
```

Ex: allpos

- **Define** `allpos`**: int list** $\rightarrow$ **bool that returns** `true` **if all elements of the list are greater than zero,** `false` **otherwise.**

- **First: To calculate** `allpos lis`**, suppose** a **=** `allpos (tl lis)`**. Calculate** `allpos lis` **from** `hd lis` **and** a**:**

- **Second: Define** `allpos`**:**

```
let rec allpos lis = match lis with
        [] ->
      | h::t ->
```

Ex: pairsums

- **Define** `pairsums`**: (int * int) list** $\rightarrow$ **int list that sums the elements of each element of its argument:**
- **E.g.** `pairsums [(3, 4); (5, 6)] = [7; 11]`**.**
- **First: To calculate** `pairsums lis`**, suppose** r **=** `pairsums (tl lis)`**. Calculate** `pairsums lis` **from** `hd lis` **and** r**:**
- **Second: Define** `pairsums`**:**

```
let rec allpos lis = match lis with
        [] ->
      | (i,j)::t ->
```

The recursion fairy *redux*

- **The recursion fairy as given above does not always work, for various reasons:**
  - `f` **may have multiple arguments**
  - **The base case may not simply be the empty list**
  - **The function may be impossible to define by recursion *directly*; an auxiliary function may be needed. (Or: it may be possible, but the resulting function is very inefficient.)**
  - **It may not be possible to define the function by simple recursion *on the tail of the list*.**

The recursion fairy *redux* (cont.)

- **A more general version of the recursion fairy would be: "To define `f x y ... z`, suppose `x'`, `y'`, ..., `z'`, are values derived from `x`, `y`, ..., `z`, that are somehow *smaller*. Suppose that, given r = `f x' y' ... z'`, you can calculate `f x y ... z` directly (i.e. non-recursively) from r, `x`, `y`, ..., `z`. Then define `f` by:**

```
let rec f x y ... z =
      if x, y, ..., z too small
      then handle base case
      else calculate result from f x' y' ... z'
```

Ex: revcumulsums

- **For this example, the empty list is not the only base case.**
- `revcumulsums lis` **is the list consisting of the sum of all the elements followed by the sum of the tail, followed by the sum of the tail of the tail, etc.:**
- `revcumulsums [1; 2; 3; 4] = [10; 9; 7; 4].`
- **First: To calculate** `revcumulsums lis`**, suppose** r = `revcumulsums (tl lis)`**, and that** `tl lis` **is not empty. Calculate** `revcumulsums lis` **from** r **and** `hd lis`**:**

Ex: revcumulsums (cont.)

- **Second: Define** `revcumulsums lis`**:**

```
let rec revcumulsums lis = match lis with
    (* handle base cases: *)



      | h::t ->
```

Ex: pairwisesums

- `pairwisesums [1; 2; 3; 4; 5; 6] = [3; 7; 11]`.
- **First: To calculate** `pairwisesums lis`, **suppose** r = `pairwisesums (tl (tl lis))`, **and** `tl lis` **is not empty. Calculate** `pairwisesums lis` **from** r, `hd lis`, **and** `hd (tl lis)`.

- **Second: Define** `pairwisesums lis` **(assume** $|$`lis`$|$ **is even):**

```
let rec pairwisesums lis = match lis with
   (* handle base cases: *)

    | h::ht::tt ->
```

Ex: pairwisesums2

- `pairwisesums2 [1; 2; 3; 4; 5] = [3; 5; 7; 9]`.
- **First: To calculate** `pairwisesums2 lis`, **suppose** r = `pairwisesums2 (tl lis)`, **and** `tl lis` **is not empty. Calculate** `pairwisesums2 lis` **from** r, `hd lis`, **and** `hd (tl lis)`.

- **Second: Define** `pairwisesums2 lis`:

```
let rec pairwisesums2 lis = match lis with
   (* handle base cases: *)


    | h::ht::tt ->
```

Ex: pairwiseequal

- `pairwiseequal [1; 2; 3] [1; 4; 3] = [true; false; true].`
- **First: Suppose** m = `pairwiseequal (tl lis1) (tl lis2).` **Calculate** `pairwiseequal lis1 lis2` **from** m, `hd lis1`, **and** `hd lis1`.

- **Second: Define** `pairwiseequal lis1 lis2`. **This requires a use of match that we haven't seen before:**

```
let rec pairwisesums2 lis = match (lis1, lis2) with
        ([], []) ->

      | (h1::t1, h2::t2) ->
```

Ex: append

- `append lis1 lis2 = lis1 @ lis2`.
- **First: Recursion is on** `lis1`. **To calculate** `append lis1 lis2`, **suppose** lis' = `append (tl lis1) lis2`. **Calculate** `append lis1 lis2` **from** lis' **and** `hd lis1`.

- **Second: Define** `append`:

```
let rec append lis1 lis2 = match lis1 with
      [] ->

     h::t ->
```

Ex: reverse

- `reverse [1;2;3] = [3;2;1].`
- **First: To calculate** `reverse lis`**, suppose** r **=** `reverse (tl lis)`**. Calculate** `reverse lis` **from** r **and** `hd lis`**. (Hint: you have to use** `@`**.)**

- **Second: Define** `reverse`**:**

```
let rec reverse lis = match lis with
      [] ->

     h::t ->
```

Ex: reverse, again

- **The problem with the previous version of** `reverse` **is that, because running time of** `lis1 @ lis2` **is** $O(|\texttt{lis1}|)$**,** `reverse` **is: _____. To obtain a more efficient version of** `reverse`**, we instead define the function** `revapp`**:**

- `revapp lis1 lis2 = (reverse lis1) @ lis2`. **E.g.** `revapp [1; 2; 3] [4; 5] = [3; 2; 1; 4; 5]`.

- **First:** `revapp lis1 lis2` **is defined by recursion on the tail of** `lis1`**. However,** `revapp (tl lis1) lis2` **is of no help. Instead, find values** `lis2'` **such that you can calculate** `revapp lis1 lis2` **directly from** `revapp (tl lis1) lis2'`**.**

  - **What is** `lis2'`**?**

- **Suppose r = `revap (tl lis1) lis2'`. How can you calculate `revapp lis1 lis2` from r?**

- **Second: Define `revapp`:**

```
let rec revapp lis1 lis2 = match lis1 with
      [] ->

     h::t ->
```

- **What is the time complexity of `rev2`?**
- **Define `reverse` using `rev2`:**

Ex: unencode

- A simple method of compressing data that is effective on some kinds of data is *run-length encoding*, where a list of values is replaced by a list of pairs, each giving a value and a number of repetitions of that value.

- In OCaml, we could encode a char list as an (int * char) list, where each pair gives the number of repetitions of the char. E.g. `[(3, 'a'); (1, 'b'); (2, 'a')]` represents the list `['a'; 'a'; 'a'; 'b'; 'a'; 'a']`.

- `unencode enc`: (char * int) list $\rightarrow$ char list takes an encoded list `enc` and returns its expanded form.

Ex: unencode

- **First: Suppose** r **=** `unencode (tl enc)`**. Calculate** `unencode enc` **as a function of** r **and** `hd enc` **(hint: you'll need an auxiliary function):**

- **Second: Define** `unencode`**:**

Ex: unencode, again

- **You can define** `unencode` **without using the auxiliary function. The trick is that the recursion is not on** `tl enc` **but on another list** `enc'` **that is related to** `enc`**. From** `unencode enc'` **you can calculate** `unencode enc` **easily: Suppose** u = `unencode enc'`**; then** `unencode enc = hd enc ::` u**.**

- **First: what is** `enc'`**?**

- **Second: Define** `unencode`**:**

Ex: encode

- `encode` **is the inverse of unencode. Its type is char list** $\rightarrow$ **(int * char) list.**
- `encode lis` **can be calculated by simple recursion on** `lis`**.**
- **First: Suppose** enc **=** `encode (tl lis)` **(assuming** `lis` **is non-empty). How can you calculated** `encode lis` **from** enc **and** `hd lis`**? (Hint: you'll need a conditional expression here.)**
- **Second: Define** `encode`**:**