Lecture 2 — OCaml basics and recursive functions on lists

- In this lecture, you will first learn some more basic OCaml, and then practice writing recursive functions on lists, which is perhaps the most common kind of programming done in functional languages. (In this course, we will also spend a lot of time writing recursive functions on trees.)

  Specifically, we'll talk about these OCaml features:

  - Let expressions and scope
  - Functions on tuples
  - Pattern-matching

  and then talk about how you can write recursive functions easily if you learn to believe in the recursion fairy.

Let expressions

- **The let expression is fundamental in OCaml because it is how names are introduced.**
- **We saw in the last class how let is used at the top level:**
  - `let x =` *`expr`*`;;`
  - `let f` *`args`* `=` *`expr`*`;;`
  - `let rec f` *`args`* `=` *`expr`*`;;`

Scope in OCaml

- "Scope" means: in what region of the program can a particular name be used?
- Scope of top-level let expressions:

`let` x `=` e`;;` — scope of x is everything that follows this let

`let` f x `=` e`;;` — scope of x is e; scope of f is everything that follows this let

`let rec` f x `=` e`;;` scope of x is e; scope of f is e and everything that follows this let

Let expressions (cont.)

- **Let expressions can also appear *within* other expressions, to introduce local names.**
- **Syntax of non-top-level (aka "nested") let expressions:**
  - `let x =` *`expr1`* `in` *`expr2`*

    — evaluate *expr1* and return value of *expr2* (which can refer to `x`)
  - `let f` *`args`* `=` *`expr1`* `in` *`expr2`*

    — define function `f` (with *expr1* as its body) and return value of *expr2* (which can call `f`)
  - `let rec f` *`args`* `=` *`expr1`* `in` *`expr2`*

    — define function `f` and return value of *expr2* (which can call `f`)

Let expressions (cont.)

- **Give the values of these expressions:**

```
(let x = 4 in x*x) + 5
```

```
let x = (let y = 1+2 in y*y) in x*x
```

```
let sumsqrs x y = let sqr a = a*a
                  in sqr x + sqr y
in sumsqrs 3 5
```

```
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))
in binom 1 1
```

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: Suppose we have a file with a series of top-level let definitions. If we replace every `;;` by `in`, the program becomes one large `let` expression; the scope of each name would be the same.**

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

- **In let expressions and function definitions, can use *patterns* instead of variables. This is handy when defining functions on structured values like tuples and lists.**
- **Here are three equivalent 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 yet 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
```

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 → 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 is too simple to always work. The proper recursion may not be simply on the tail of the list, and the base cases may include more than the empty list. And what if there is more than one argument?**

- **We won't try to give a completely general definition. But the general idea is always the same:** *Make a recursive call using arguments that are, in some way, "smaller" then the arguments you're given.* **Assume** *the result you get back is correct, and go from there.*

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: 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: 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`: (char * int) list → char list takes an encoded list `enc` and returns its expanded form.

Ex: unencode

- **First: Suppose** `hd enc` **is (1, x) and r =** `unencode (tl enc)`**. Calculate** `unencode enc` **as a function of r and x.**

- **Second: Suppose** `hd enc` **is (n, x), where $n > 1$, and r =** `unencode` $(n-1,\ x)$ `:: (tl enc)`**. Calculate** `unencode enc` **as a function of r and x.**

- **The previous two questions suggest that, for** `unencode`**, the trick is making the correct recursive call, depending upon ...**

```
let rec unencode lis =
```

Wrap-up

- **Today we discussed:**
  - **More OCaml — `let`, patterns, `match`, lists**
  - **The recursion fairy**
- **We discussed it because:**
  - **Writing recursive functions on lists is an essential skill in functional programming.**
- **In the next class, we will:**
  - **Talk about how to define new types in OCaml, esp. trees**
  - **Talk about *abstract syntax tree***
- **What to do now:**
  - **MP1**
  - **For more on today's topic, read the supplementary notes on the web.**