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.**

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.)

- **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

    [] ->
```

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 ->
```

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`**:**