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:**

```
let fst_of_3 (x, y, z) = x
let fst_of_3 t = let (x, y, z) = t
                 in x
```

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:**

  sum1 3 4 sum2 (3, 4)

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

  sum1 : int → int → int sum2: int * int → int

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

  Type error

- `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 → 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]`**:**

let rev2 (x :: y :: t) = y :: x :: t

Functions on lists (cont.)

- **Ex: second: int list → 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 0
                 else if tl lis = [] then hd lis
                                     else hd (tl lis)

let second lis = match lis with
    [] -> 0
  | [h] -> h
  | h::ht::tt -> ht
```

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`**?**

hd lis + s

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

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

  hd lis > 0 && a

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

```
let rec allpos lis = match lis with
        [] -> true
      | h::t -> h>0 && allpos 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**:**

  (fst (hd lis) + snd (hd lis)) :: r

- **Second: Define** `pairsums`**:**

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

(hd lis + hd r) :: r

Ex: revcumulsums (cont.)

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

```
let rec revcumulsums lis = match lis with
    (* handle base cases: *)
        [] -> []
      | [h] -> [h]
      | h::t -> let r = revcumulsums t
                in (h + hd r) :: r
```

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)`.

  (hd lis + hd (tl lis)) :: r

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

```
let rec pairwisesums lis = match lis with
    (* handle base cases: *)
      [] -> []
    | h::ht::tt -> (h+ht) :: pairwisesums 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)`.

  (hd lis + hd (tl lis)) :: r

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

```
let rec pairwisesums2 lis = match lis with
   (* handle base cases: *)
      [] -> []
    | [h] -> h
    | h::ht::tt -> (h+ht) :: pairwisesums2
                             (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`.

  (hd lis1 = hd lis2) :: m

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

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

      | (h1::t1, h2::t2) -> (h1=h2) ::
                            pairwiseequal t1 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`**.**

  hd lis1 :: lis'

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

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

      h::t -> h :: append t lis2
```

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** `@`**.)**

  r @ [hd lis]

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

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

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: $O(|lis1|^2)$. 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'`? hd lis1 :: lis2

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

  It is just r

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

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

      h::t -> revapp t (h :: lis2)
```

- **What is the time complexity of** `rev2`**?** $O(|lis1|)$
- **Define** `reverse` **using** `rev2`**:**

  let reverse lis = revapp lis []

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):**

```
let rec rpt i x = if i=0 then []
                  else x :: rpt (i-1) x
in let (a,b) = hd enc
   in (rpt a b) @ r
```

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

```
let rec unencode lis = match lis with
    [] -> []
  | h::t -> let (a,b) = h
            in (rpt a b) @ unencode t
```

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` = (snd `hd enc`) :: u.

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

  (fst (hd enc) - 1, snd (hd enc)) :: tl enc

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

```
let rec unencode lis = match lis with
    [] -> []
  | (i,x)::t -> if i=0 then unencode t
                else x :: unencode ((i-1,x)::t)
```

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.)**

```
if hd lis = hd (tl lis)
then (fst (hd enc)+1, hd lis) :: tl enc
else (1, hd lis) :: enc
```

- **Second: Define** `encode`:

```
let rec encode lis = match lis with
  [] -> [] | [x] -> [(1, x)] |
  h::t -> let (a,b)::r = encode t
          in if h=b then (a+1, b) :: r
             else (1,h) :: (a,b) :: r
```