A bit more on tuples

```
utop # (1, 'w');;
- : int * char = (1, 'w')
utop # (1, 'w', "");;
- : int * char * string = (1, 'w', "")
utop # (1, ('w', ""));;
- : int * (char * string) = (1, ('w', ""))
utop # (1, 'w', "") = (1, ('w', "")) ;;
Error: This expression has type 'a * 'b but an
expression was expected of type int * char *
string
utop # ();;
- : unit = ()
utop # 1;;
- : int = 1
```

pair

triple

pair of value and pair

“dummy value”

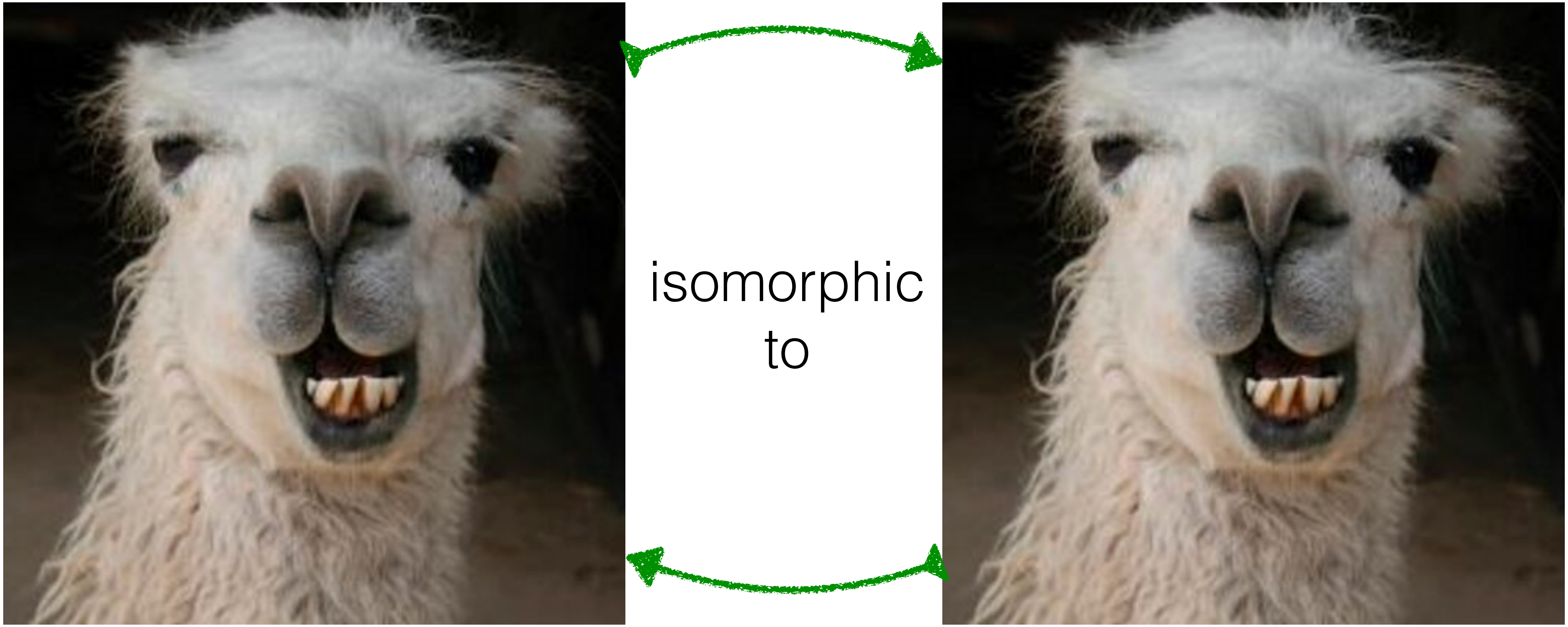
“empty” tuple

“singleton” tuple

“Isomorphism”

isos = equal
morphism = shape

Two data types are said to be *isomorphic* if they can be converted to each other without loss of information.



isomorphic
to

.BMP

.PNG

.BMP

not
isomorphic
to

.JPG

Obs: 'a * 'b * 'c ≈ 'a * ('b * 'c)

```
utop # let f (x, y, z) = (x, (y, z));;
val f : 'a * 'b * 'c → 'a * ('b * 'c) = <fun>

utop # let g (x, (y, z)) = (x, y, z);;
val g : 'a * ('b * 'c) → 'a * 'b * 'c = <fun>

utop # (1, 2, 3) ▷ f ▷ g;;
- : int * int * int = (1, 2, 3)

utop # (1, (2, 3)) ▷ g ▷ f;;
- : int * (int * int) = (1, (2, 3))
```

Lists

EFC :: W4L1

Dan R. Ghica
EFC

Quick recap of syntax

```
let x = … ;;

let x = … in …

fun x → …

let f x = …

let f (x, y) = …

let f x y = …

match … with | … → … | … → …

let f = function … → … | … → …
```

- Find the largest of 2
- Find the largest of 10
- Find the largest of 1,000,000,000

tuples not an option!

What if we could **define** “tuples of any size”?

- what **is** a “tuple of any size”?
- it can be the **empty** tuple … `() : unit`
- it can be a **singleton** … `x : 'a  ≈(‘a * unit)`
- it can be a **pair** … `(x, y) :`

  `’a*'a ≈(‘a * singleton)`
- it can be a **triple** …`(x,y,z):`

  `’a*’a*'a≈('a * pair)`
- n+1-tuple … `≈('a * ntuple)`

A tuple of arbitrary length is empty or…

… a pair of something with a tuple of arbitrary length.

list = n-tuple = Empty
list = n-tuple = int * n-tuple

```
type intlist = Empty
             | Cons of (int * intlist)
```

```
type 'a list = Empty
             | Cons of (int * 'a list)
```

```
[]        : 'a list
x :: xs   : 'a list
```

Notation for lists

No computation involved.

```
[1; 2; 3; 4] =
1 :: [2; 3; 4] =
1 :: (2 :: [3; 4]) =
1 :: (2 :: (3 :: [4])) =
1 :: 2 :: 3 :: 4 :: []
```

Pattern matching for lists

```
match xs with
  | [] → …
    (* Empty *)
  | x :: xs → …
    (* Cons (x, xs) *)
```

Example 1: **hd**

```
let hd xs = match xs with
  | []      → failwith ”hd”
  | x :: xs → x
```

Example 1: **hd**

```
let hd = function
  | []      → failwith ”hd”
  | x :: xs → x
```

Example 1: **hd**

```
let hd = function
  | []     →  failwith ”hd”
  | x :: _ →  x
```

Example 2: **tl**

```
let tl = function
    | []        → failwith ”tl”
    | _ :: xs → xs
```

Alternative: **hd, tl**

```
let hd’ = function x :: _  → x
let tl’ = function _ :: xs → xs
```

```
# let hd' = function x :: _ → x;;
Warning 8: this pattern-matching is not
exhaustive.
Here is an example of a value that is not matched:
[]
val hd' : 'a list → 'a = <fun>
# hd [1;2;3];;
- : int = 1
# hd' [1;2;3];;
- : int = 1
# hd [] ;;
Exception: Failure "hd".
# hd' [] ;;
Exception: Match_failure ("//toplevel//", 2, -24).
```

More on pattern-matching

```
match xs with
| xs → … anything
| [] → … the empty list
| x :: xs → … a non-empty list
| x :: x’ :: xs → … at least 2 elms
| x :: 3 :: xs → … 2nd elem is 3
| [1;x;y] → … 3 elems, first one is 1
| 1 :: (2,3) :: xs → … impossible!
```

Which are the odd ones out?

`[1; 3; 5] =`

A. `1 :: 3 :: [5]` ✅

B. `1 :: 3 :: 5 :: []` ✅

~~C. `1 :: 3 :: 5`~~

~~D. `[1; 3] :: [5]`~~

~~E. `[1] :: [3; 5]`~~

What patterns match?

```
match [1; 3; 5] with
```

~~A. [] ?~~

B. [x; y; z] ? ✅

C. x :: [y; z] ? ✅

D. x :: y :: x :: xs ? ✅

~~E. 1 :: x :: 3 :: xs ?~~

Main list operations

- http://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html
- `List.hd;;`
- `open List;;`

```
val length : 'a list → int
```

```
list [1; 3; 5] = 3
```

```
val nth : 'a list → int → 'a
```

```
nth [1; 3; 5] 0 = 1

nth [1;2;3] 4;;
Exception: Failure "nth".
```

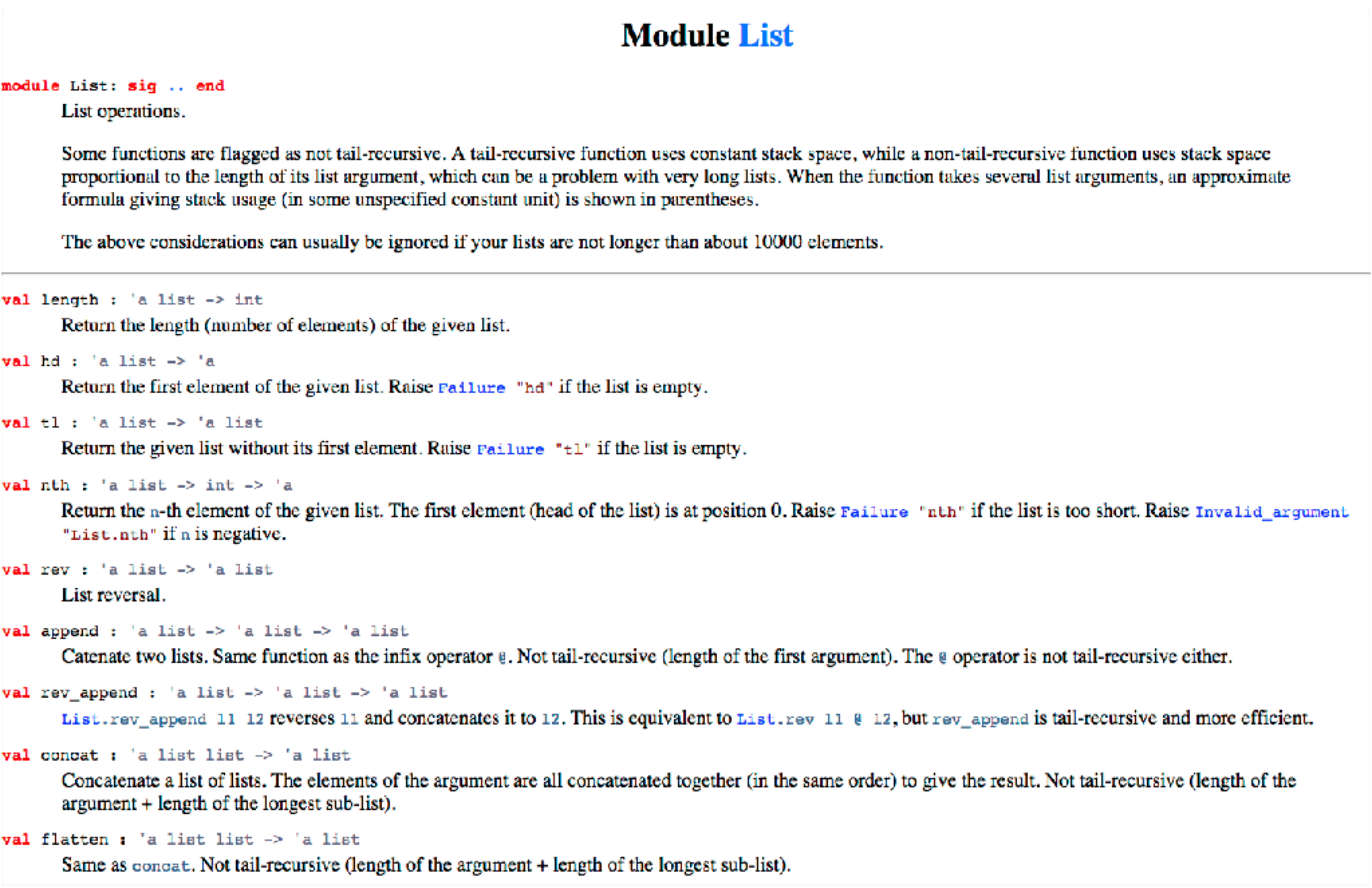
```
val rev : 'a list → 'a list
```

```
rev [1; 3; 5] = [5; 3; 1]
```

```
val append
: 'a list → 'a list → 'a list
```

```
append [1; 3; 5] [7; 9] = [1; 3; 5; 7; 9]

[1; 3; 5] @ [7; 9] = [1; 3; 5; 7; 9]
```



Module List

`module List: sig .. end`

List operations.

Some functions are flagged as not tail-recursive. A tail-recursive function uses constant stack space, while a non-tail-recursive function uses stack space proportional to the length of its list argument, which can be a problem with very long lists. When the function takes several list arguments, an approximate formula giving stack usage (in some unspecified constant unit) is shown in parentheses.

The above considerations can usually be ignored if your lists are not longer than about 10000 elements.

`val length : 'a list -> int`

Return the length (number of elements) of the given list.

`val hd : 'a list -> 'a`

Return the first element of the given list. Raise `Failure "hd"` if the list is empty.

`val tl : 'a list -> 'a list`

Return the given list without its first element. Raise `Failure "tl"` if the list is empty.

`val nth : 'a list -> int -> 'a`

Return the `n`-th element of the given list. The first element (head of the list) is at position 0. Raise `Failure "nth"` if the list is too short. Raise `Invalid_argument "List.nth"` if `n` is negative.

`val rev : 'a list -> 'a list`

List reversal.

`val append : 'a list -> 'a list -> 'a list`

Catenate two lists. Same function as the infix operator `@`. Not tail-recursive (length of the first argument). The `@` operator is not tail-recursive either.

`val rev_append : 'a list -> 'a list -> 'a list`

`List.rev_append l1 l2` reverses `l1` and concatenates it to `l2`. This is equivalent to `List.rev l1 @ l2`, but `rev_append` is tail-recursive and more efficient.

`val concat : 'a list list -> 'a list`

Concatenate a list of lists. The elements of the argument are all concatenated together (in the same order) to give the result. Not tail-recursive (length of the argument + length of the longest sub-list).

`val flatten : 'a list list -> 'a list`

Same as `concat`. Not tail-recursive (length of the argument + length of the longest sub-list).

Recursion

Recursion

```
let length = function
| [] → 0
| _::[] → 1
| _::_::[] → 2
| _::_::_::[] → 3
| _::_::_::_::[] → 4
| ???
```

Recursion:
defining a **function** in terms of how **it** operates on **smaller** data

self-reference

avoid infinite regress

Recursion on lists

```
let rec length = function
| []       → ?? (* the base *)
| x :: xs  → ?? (* the step *)
```

the head

the tail (a smaller list)

Recursion on lists

```
let rec length = function
| []      -> 0  (* the base *)
| x :: xs -> ?? (* the step *)
```

Recursion on lists

```
let rec length = function
| []      →  0           (* the base *)
| _ :: xs →  …length xs… (* the step *)
```

Recursion on lists

```
let rec length = function
| []      → 0           (* the base *)
| _ :: xs → 1+length xs (* the step *)
```

Example

```
length [5;7;9] =
length 5 :: [7;9] =
1 + length [7;9] =
1 + length 7::[9] =
1 + 1 + length [9] =
2 + length 9 :: [] =
2 + 1 + length [] =
3 + 0 =
3
```