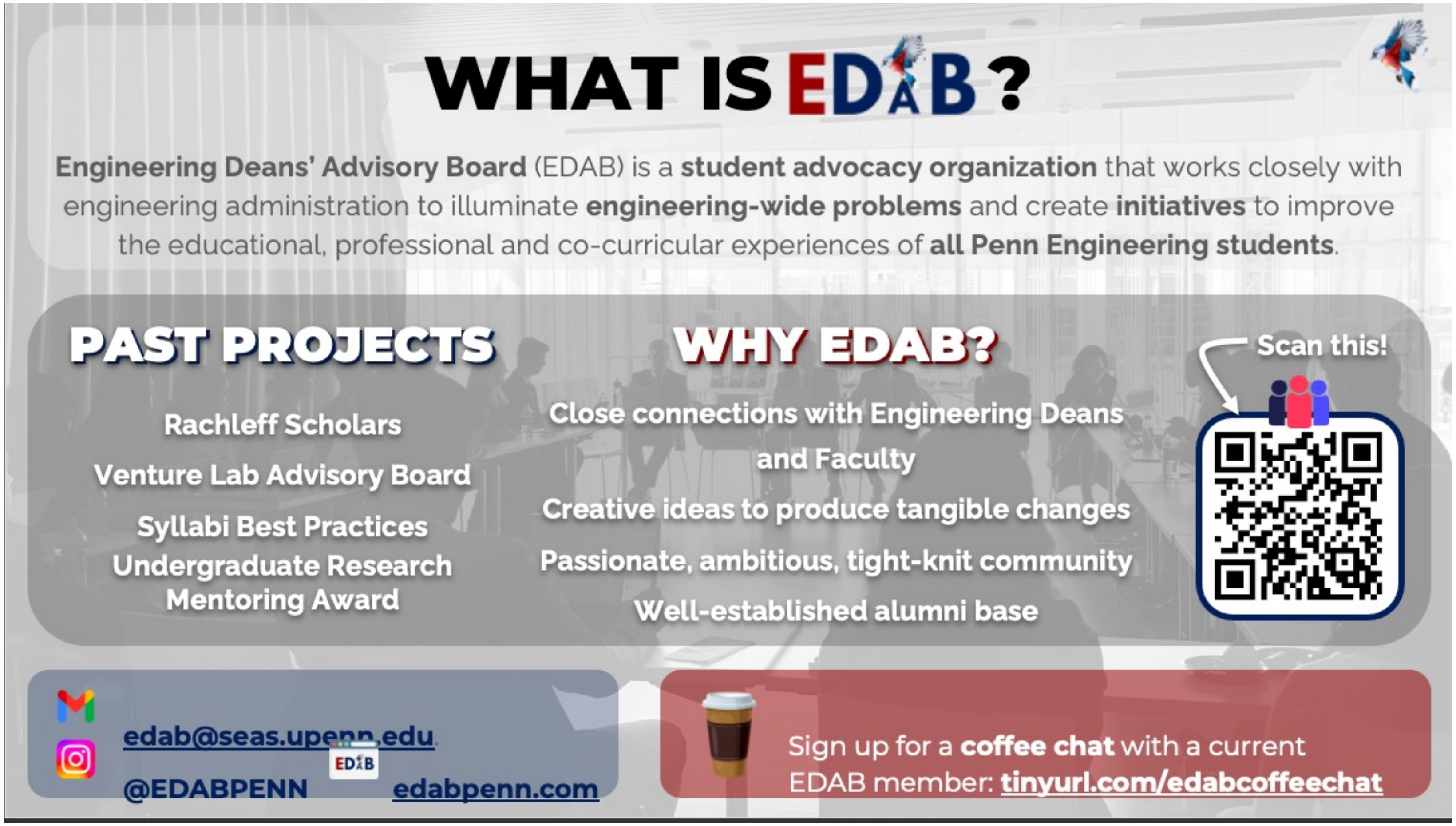
Programming Languages and Techniques (CIS1200)

Lecture 4

Lists, Recursion, and Tuples



WHAT IS EDAB?

Engineering Deans' Advisory Board (EDAB) is a **student advocacy organization** that works closely with engineering administration to illuminate **engineering-wide problems** and create **initiatives** to improve the educational, professional and co-curricular experiences of **all Penn Engineering students**.

PAST PROJECTS

Rachleff Scholars

Venture Lab Advisory Board

Syllabi Best Practices

Undergraduate Research Mentoring Award

WHY EDAB?

Close connections with Engineering Deans and Faculty

Creative ideas to produce tangible changes

Passionate, ambitious, tight-knit community

Well-established alumni base

Scan this!

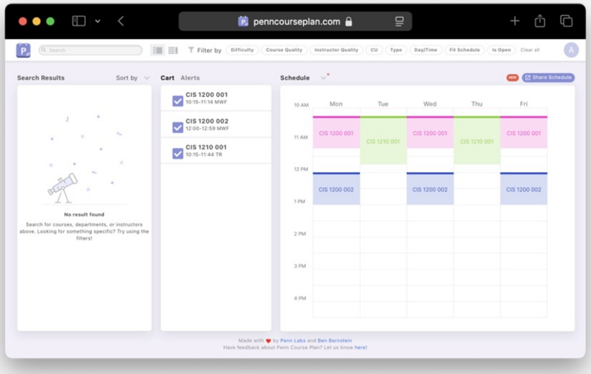
edab@seas.upenn.edu

@EDABPENN

edabpenn.com

Sign up for a **coffee chat** with a current EDAB member: tinyurl.com/edabcoffeechat

Penn Labs



No result found

Search for courses, departments, or instructors above. Looking for something specific? Try using the filters!

☑ CIS 1200 001 10:15-11:14 MWF

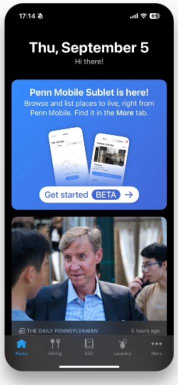
☑ CIS 1200 002 12:00-12:59 MWF

☑ CIS 1210 001 10:15-11:44 TR

Made with ♥ by Penn Labs and Ben Bernstein

Have feedback about Penn Course Plan? Let us know here!

Penn Courses



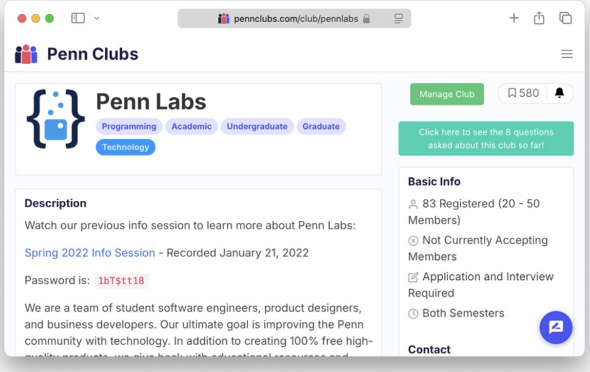
Thu, September 5

Hi there!

Penn Mobile Sublet is here!

Browse and list places to live, right from Penn Mobile. Find it in the **More** tab.

Penn Mobile



Penn Labs

Click here to see the 8 questions asked about this club so far!

Description

Watch our previous info session to learn more about Penn Labs:

Spring 2022 Info Session - Recorded January 21, 2022

Password is: `1bT$tt18`

We are a team of student software engineers, product designers, and business developers. Our ultimate goal is improving the Penn community with technology. In addition to creating 100% free high-

Basic Info

83 Registered (20 - 50 Members)

Not Currently Accepting Members

Application and Interview Required

Both Semesters

Contact

Penn Clubs

Penn Labs

Info session **today at 7:30 PM**
Huntsman F65

Applications **open tonight**
Due Monday, 9/9

pennlabs.org/apply

Subscribe
for updates

CIS 1200 Announcements

- Homework 1: OCaml Finger Exercises
  - Due: Tuesday at 11.59pm ET
  - Must submit via course website
  - Use the ’Zip’ option in the ‘Run Submission’ menu *not* Codio’s “export as zipfile”

- Read Chapters 3 (Lists) and 4 (Tuples) of the lecture notes

- We will start Chapters 5 & 6 on Monday

Review: What is a list?

A list is either:

`[]` the *empty* list, sometimes called *nil*

or

`v :: tail` a *head* value v, followed by a list of the remaining elements, the *tail*

- `list` an example of an *inductive datatype*
- We inspect a list value by *pattern matching* against its shape
- The natural way to process a list is with *structural recursion*

Calculating with Matches

- Consider how to evaluate a match expression:

```
begin match [1;2;3] with
  | [] -> 42
  | first::rest -> first + 10
end
```

↦

1 + 10

↦

11

Note: `[1;2;3]` means `1::(2::(3::[]))`

It doesn’t match the pattern [], so the first branch is skipped, but it *does* match the pattern `first::rest` when `first` is `1` and `rest` is `(2::(3::[]))`.
So we substitute 1 for `first` in the second branch.

Recursion

The Inductive Nature of Lists

A list value is either:

`[]` the *empty* list, sometimes called *nil*

or

`v :: tail` a *head* value v, followed by a list value containing the remaining elements, the *tail*

- Why is this well-defined? The definition of list mentions ‘list’!
- Answer: ‘list’ is *inductive*:
  - The empty list [] is the (only) list of 0 elements
  - To construct a list of n+1 elements, add a head element to an *existing* list of n elements
  - The set of list values contains *all and only* values constructed this way
- Corresponding computation principle: *recursion*

Recursion

Recursion principle:

Compute a function value for a given input by combining the results for strictly smaller parts of the input.

- The *recursive* structure of the computation follows the *inductive* structure of the input.

- Example:

```
length (1::2::3::[])  =  1 + length (2::3::[])
length (2::3::[])     =  1 + length (3::[])
length (3::[])        =  1 + length []
length []             =  0
```

Recursion Over Lists

The function calls itself *recursively* so the function declaration must be marked with rec.

Lists are either empty or nonempty. *Pattern matching* determines which.

```
let rec length (l : string list) : int =
  begin match l with
  | [] -> 0
  | ( x :: rest ) -> 1 + length rest
  end
```

If the list is non-empty, then “x” is the first string in the list and “rest” is the remainder of the list.

Calculating with pattern matching and recursion

Calculating with Recursion

```
length ["a"; "b"]
```

⟼ *(substitute the list for l in the function body)*

```
begin match "a"::"b"::[] with
| [] -> 0
| ( x :: rest ) -> 1 + length rest
end
```

⟼ *(second case matches with rest =* "b"::[]*)*

```
1 + length ("b"::[])
```

⟼ *(substitute the list for l in the function body)*

```
1 + begin match "b"::[] with
        | [] -> 0
        | ( x :: rest ) -> 1 + length rest
        end
```

⟼ *(second case matches again, with rest = [])*

```
1 + (1 + length [])
```

⟼ *(substitute [] for l in the function body)*

…

⟼ 1 + 1 + 0 ⇒ 2

```
let rec length (l:string list) : int=
  begin match l with
  | [] -> 0
  | ( x :: rest ) -> 1 + length rest
  end
```

More recursion examples…

```
let rec sum (l : int list) : int =
  begin match l with
  | [] -> 0
  | ( x :: rest ) -> x + sum rest
  end
```

```
let rec contains (l:string list) (s:string):bool =
  begin match l with
  | [] -> false
  | ( x :: rest ) -> s = x || contains rest s
  end
```

4: What best describes the behavior of (foo 3 l) ? It returns true if...

✓ 0

1. Every element of l is less than 3.

0%

2. Every element of l is greater than 3

0%

3. There exists an element in l that is less than 3

0%

4. There exists an element in l that is greater than 3

0%

What best describes the behavior of the function call `(foo 3 l)` ?
It returns true if…

```
let rec foo (z:int) (l : int list): bool =
  begin match l with
  | [] -> true
  | ( x :: rest ) ->
    (x > z) && foo z rest
  end
```

Answer: every element is greater than 3

The General Pattern: Structural Recursion Over Lists

Structural recursion builds an answer from smaller components:

```
let rec f (l : … list) … : … =
  begin match l with
  | [] -> …                   (* BASE CASE *)
  | ( hd :: rest ) ->
         … (f rest) …         (* INDUCTIVE CASE *)
  end
```

The branch for `[]` calculates the value (`f []`) directly.

- this is the *base case* of the recursion

The branch for `hd::rest` calculates
(`f (hd::rest)`) given `hd` and (`f rest`).

- this is the *inductive case* of the recursion

Tuples and Tuple Patterns

Two Forms of Structured Data

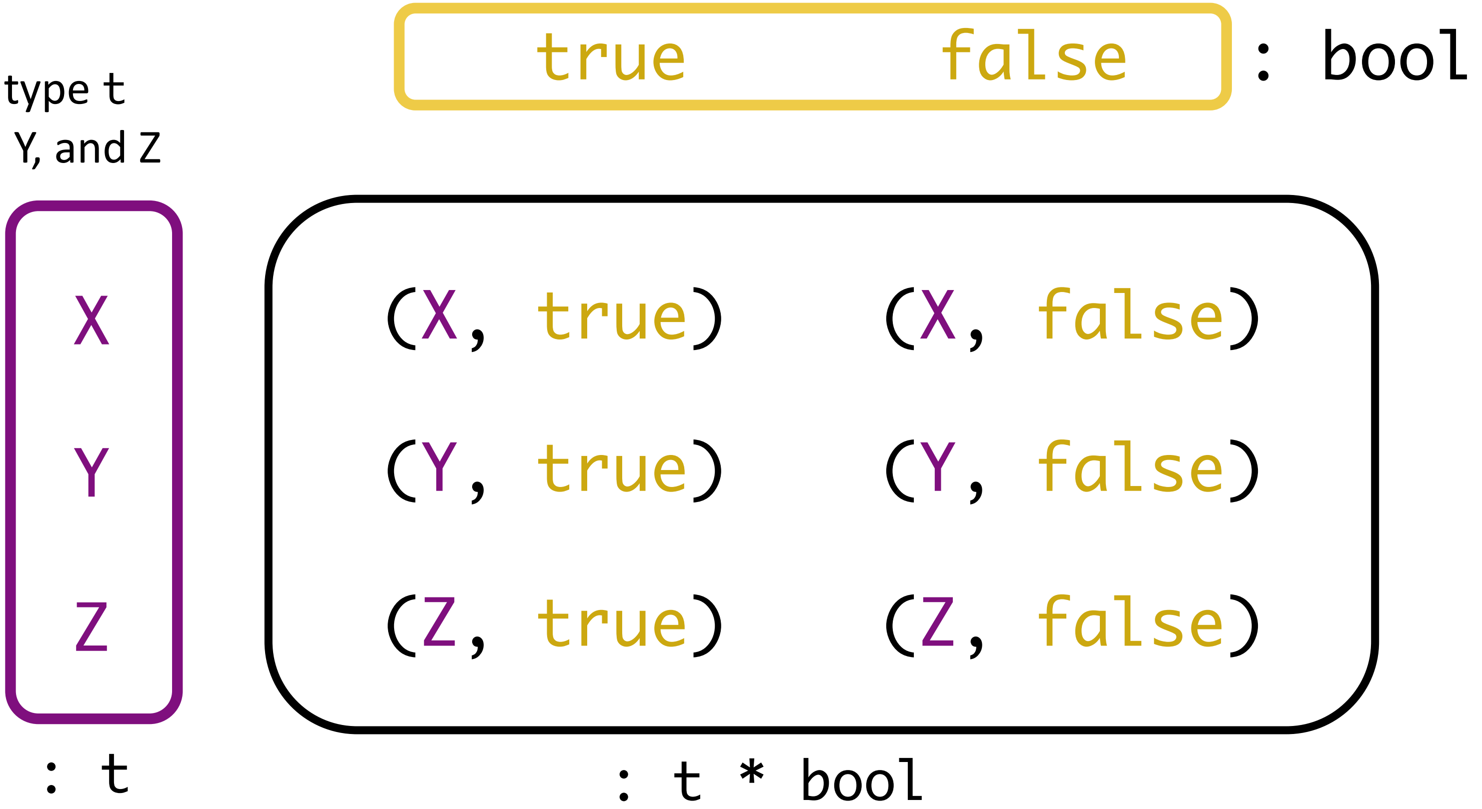
OCaml provides two basic ways of packaging multiple values together into a single compound value:

- **Lists:**
  - *arbitrary-length* sequence of values of a *single type*
  - example: a list of email addresses
- **Tuples:**
  - *fixed-length* sequence of values, possibly of *different types*
  - example: tuple of name, phone #, and email

(Cartesian) Products

- The values of a *tuple* (or *product*) *type* are tuples of values from each component type.

suppose the type t
has values X, Y, and Z



true false : bool

X
Y
Z
: t

(X, true) (X, false)
(Y, true) (Y, false)
(Z, true) (Z, false)
: t * bool

The tuple type `t * bool` has all *pairs* of values

Tuples

- In OCaml, tuple *values* are created by writing a sequence of expressions, separated by commas, inside parens:

```
let my_pair = (3, true)
let my_triple = (“Hello”, 5, false)
let my_quadruple = (1, 2, ”three”, false)
```

- Tuple *types* are written using infix ‘*’
  - e.g., `my_triple` has type:

```
string * int * bool
```

Pattern Matching on Tuples

- Tuples can be inspected by pattern matching:

```
let first (x: string * int) : string =
  begin match x with
  | (left, right) -> left
  end

first (“b”, 10)
⇒
“b”
```

- As with lists, tuple patterns follow the syntax of tuple values and give names to the subcomponents so they can be used on the right-hand side of the -> in each case

Mixing Tuples and Lists

- Tuples and lists can mix freely:

```
[(1,"a"); (2,"b"); (3,"c")]
             : (int * string) list
```

```
([1;2;3], ["a"; "b"; "c"])
             : (int list) * (string list)
```

Nested Patterns

- We’re seen several kinds of *simple patterns*:

`[]`	*matches empty list*
`x::tl`	*matches nonempty list*
`(a,b)`	*matches pairs (tuples with 2 elts)*
`(a,b,c)`	*matches triples (tuples with 3 elts)*

- We can build *nested patterns* out of simple ones:

`x :: []`	*matches lists with exactly 1 element*
`[x]`	*matches lists with exactly 1 element*
`x::(y::tl)`	*matches lists with at least 2 elements*
`(x::xs, y::ys)`	*matches pairs of non-empty lists*

Wildcard Pattern

- Another handy simple pattern is the wildcard "_"

- A wildcard pattern indicates that the value of the is not used on the right-hand side of the match case
  - And hence needs no name

`_::tl` *matches a non-empty list, but only names its tail*

`(_,x)` *matches a pair (2-tuple), but only names the 2nd part*

Unused Branches

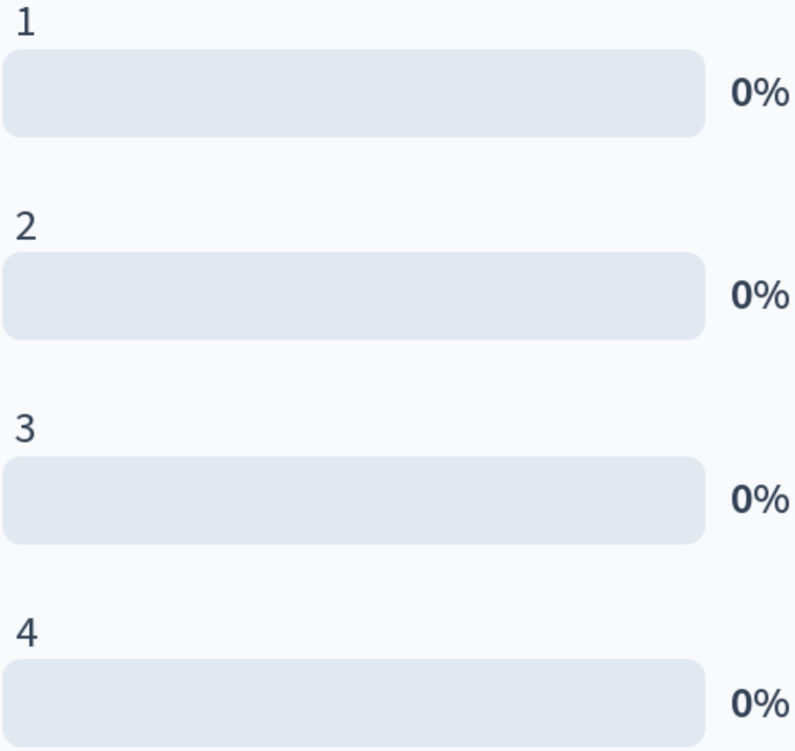
- The branches in a match expression are considered in order from top to bottom
- If you have *redundant* matches, then later branches are not reachable
  - OCaml will give you a warning in this case

```
let bad_cases (l : int list) : int =
  begin match l with
  | [] -> 0
  | x::_ -> x
  | x::y::tl -> x + y     (* unreachable *)
  end
```

This case matches more lists than that one does

4: What is the value of this expression?

✓ 0



Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

What is the value of this expression?

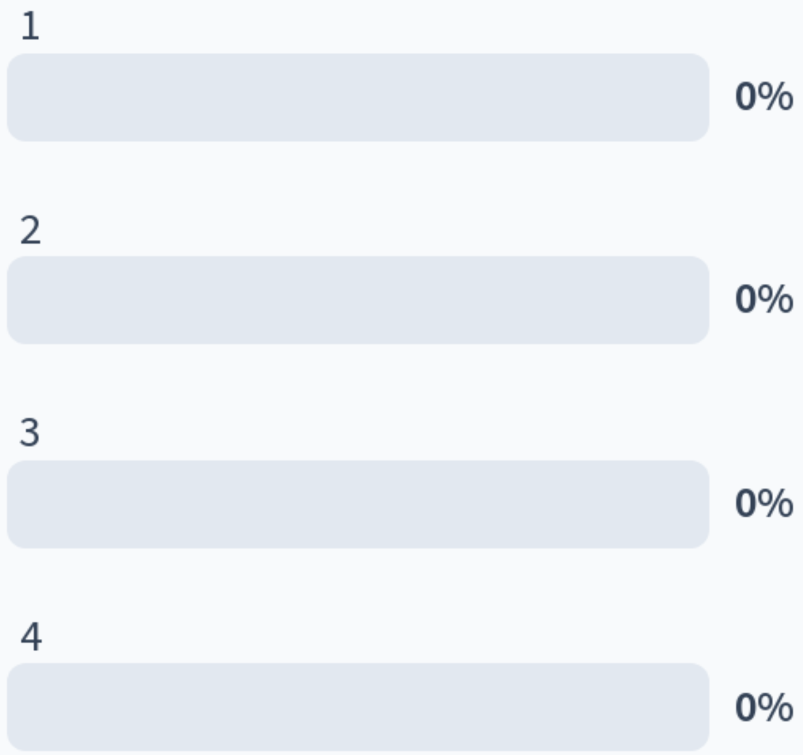
```
let l = [1; 2] in

begin match l with
   | x :: y :: t  -> 1
   | x :: []      -> 2
   | x :: t       -> 3
   | []           -> 4
end
```

Answer: 1

4: What is the value of this expression?

✓ 0



What is the value of this expression?

```
let l = [(2,true); (3,false)] in

begin match l with
   | (x,false) :: tl      -> 1
   | w :: (x,y) :: z      -> x
   | x                    -> 4
end
```

Answer: 3

Exhaustiveness

- A pattern match is said to be *exhaustive* if it includes a pattern for every possible value
- Example of a *non-exhaustive* match:

```
let sum_two (l : int list) : int =
  begin match l with
  | x::y::_ -> x+y
  end
```

- OCaml will give you a warning and show an example of what isn’t covered by your patterns

Exhaustiveness

- Example of an *exhaustive* match:

```
let sum_two (l : int list) : int =
  begin match l with
  | x::y::_ -> x+y
  | _ -> failwith "length less than 2"
  end
```

- The wildcard pattern and `failwith` eliminate the warning and make your intention explicit

Pattern Matching in let

- OCaml's `let x = e in …` notation can bind a pattern instead of a single variable:

```
let (x, y) = (true,"abc") in …
```

- Very useful for naming tuple components
- Should avoid using when the pattern is not exhaustive (i.e., there are multiple cases)
  - that is what match is for

More List & Tuple Programming

see patterns.ml

Example: zip

- zip takes two lists of the same length and returns a single list of pairs:

```
zip [1; 2; 3] ["a"; "b"; "c"] ⇒
  [(1,"a"); (2,"b"); (3,"c")]
```

```
let rec zip (l1: int list)
            (l2: string list) : (int * string) list =
  begin match (l1, l2) with
  | ([], []) -> []
  | (x:: xs, y:: ys) -> (x, y) :: (zip xs ys)
  | _ -> []
  end
```