# Programming Languages and Techniques (CIS120)

Lecture 4

Lists, Recursion and Tuples

# Announcements

- Read Chapters 3 (Lists) and 4 (Tuples) of the lecture notes
- We will start Chapters 5 & 6 on Monday
- HW01 due Tuesday at midnight
- Poll Everywhere attendance starts 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*

- The '`::`' operator, pronounced "cons", *constructs* a new list from a head element and a shorter list.
- Lists are an example of an *inductive datatype*.

# 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]` equals `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, substitute 1 for `first` in the second branch.

# The Inductive Nature of Lists

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*

- What is going on!? The definition of list mentions 'list'!
- Insight: 'list' is *inductive*:
  - The empty list [] is the (only) list of 0 elements
  - To construct a list of (1+n) elements, add an 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 subcomponents of the input.

– The 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 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 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
```

# What best describes the behavior of (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
```

1. Every element of l is less than 3.
2. Every element of l is greater than 3
3. There exists an element in l that is less than 3
4. There exists an element in l that is greater than 3

What best describes the behavior of the function call `(foo 3 l)` ?

ANSWER: every element is greater than 3

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

# 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

# Design Pattern for Recursion

1. Understand the problem

   What are the relevant concepts and how do they relate?
2. Formalize the interface

   How should the program interact with its environment?
3. Write test cases
   - If the main input to the program is an immutable list, make sure the tests cover both empty and non-empty cases
4. Implement the required behavior
   - If the main input to the program is an immutable list, look for a recursive solution…
     - Is there a direct solution for the empty list?
     - Suppose someone has given us a partial solution that works for lists up to a certain size. Can we use it to build a better solution that works for lists that are one element larger?

# 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

# Tuples

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

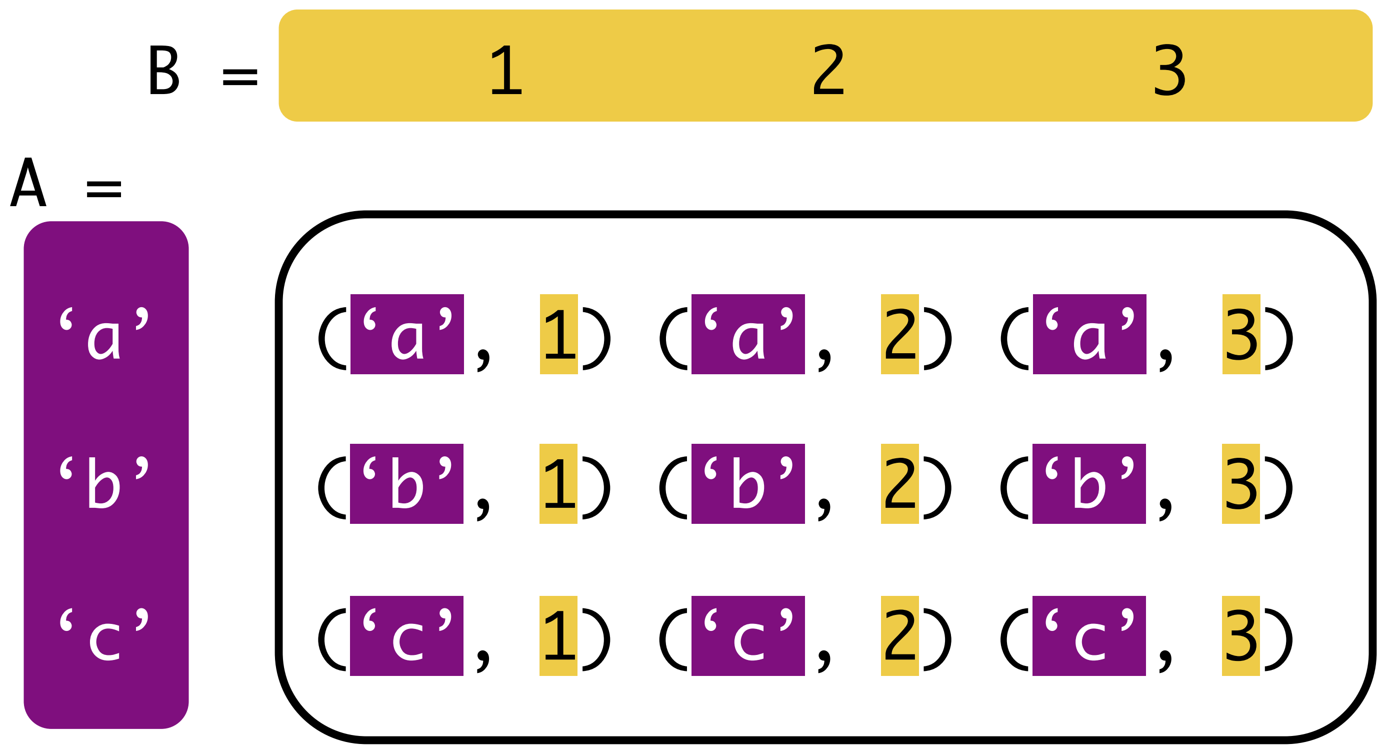
```
let my_pair = (3, true)
let my_triple = (“Hello”, 5, false)
let my_quadruple = (1, 2, ”three”, false)
```

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

```
string * int * bool
```

# Cartesian Products

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




Ocaml notation:
A * B

# 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 ->

# 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 1 element* |
| `[x]` | *matches lists with 1 element* |
| `x::(y::tl)` | *matches lists of length at least 2* |
| `(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 corresponding subcomponent 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 tail*

`(_,x)` *matches a pair, 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 some later branches might not be 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
```

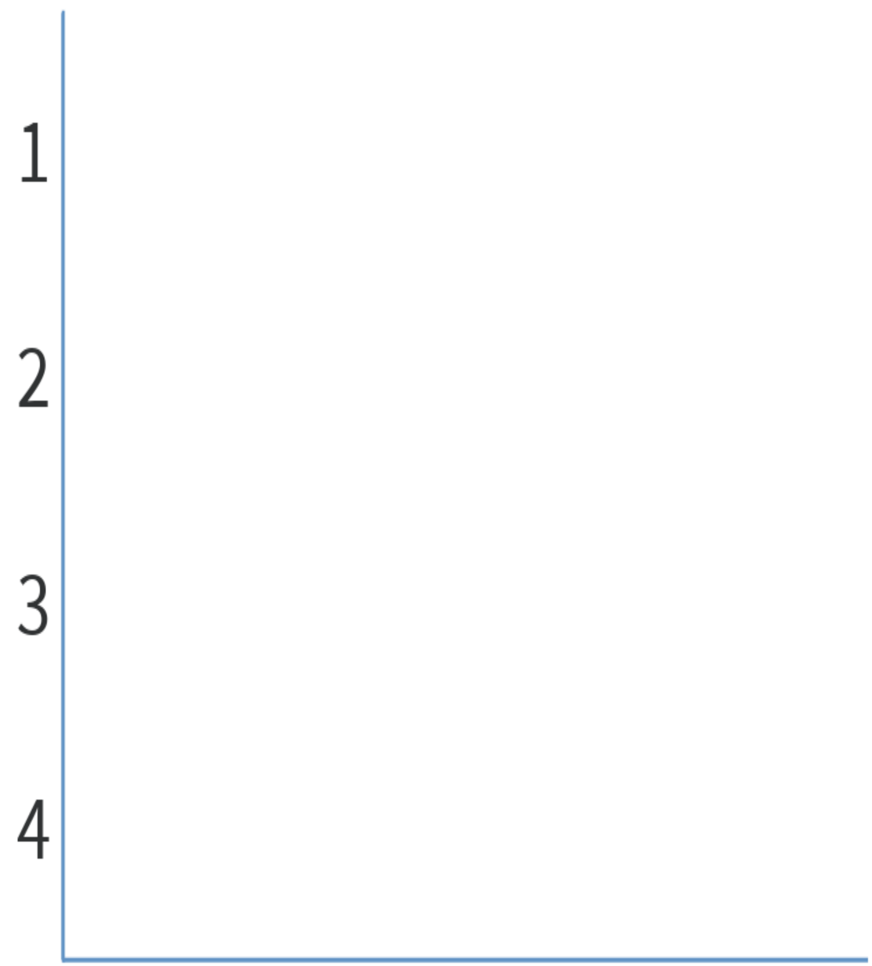
This case matches more lists than that one does.

# What is the value of this expression?

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
```



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

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

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

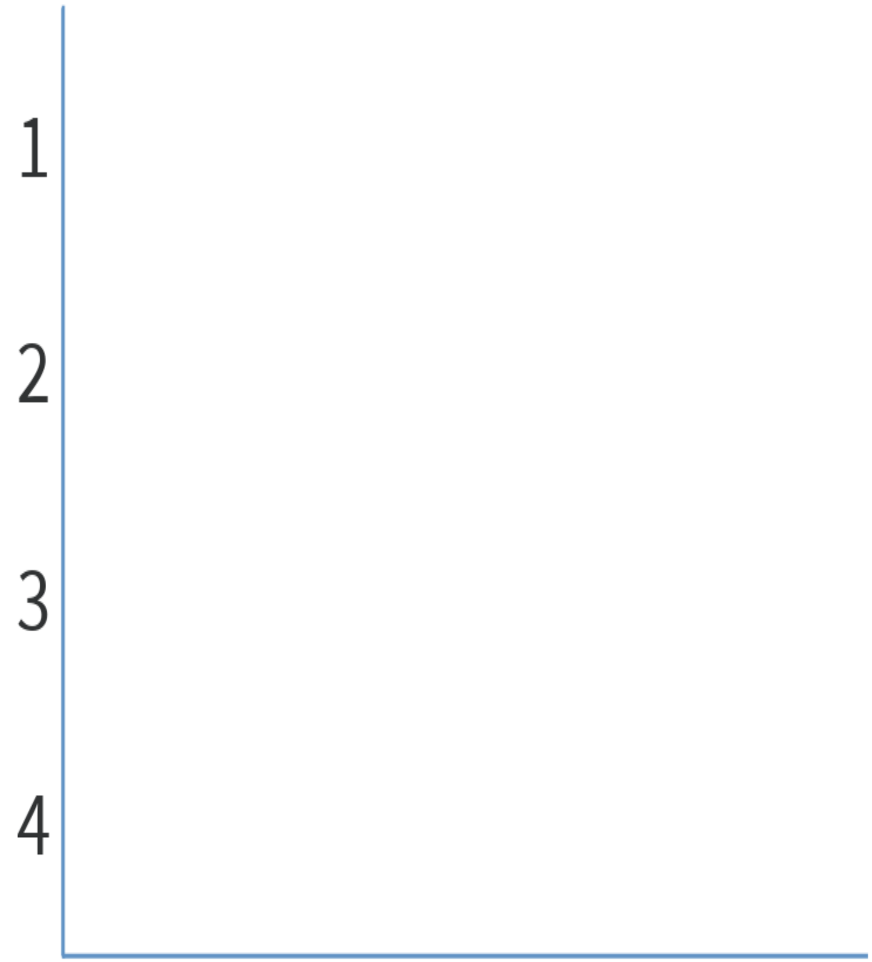
```
1
```

# What is the value of this expression?

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
```



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

What is the value of this expression?

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

begin match l with
   | (_,false) :: _        -> 1
   | _ :: (x,_) :: _       -> x
   | _                     -> 4
end
```

Answer: 3

# Exhaustive Matches

- Pattern matching is *exhaustive* if there is 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 cases

# Exhaustive Matches

- Pattern matching is *exhaustive* if there is a pattern for every possible value
- Example of an *exhaustive* match:

```
let sum_two (l : int list) : int =
  begin match l with
  | x::y::_ -> x+y
  | _ -> failwith "not a length 2 list"
  end
```

- The wildcard pattern and `failwith` are useful tools for ensuring match coverage

# 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)
  | _ -> failwith "zip: unequal length lists"
  end
```