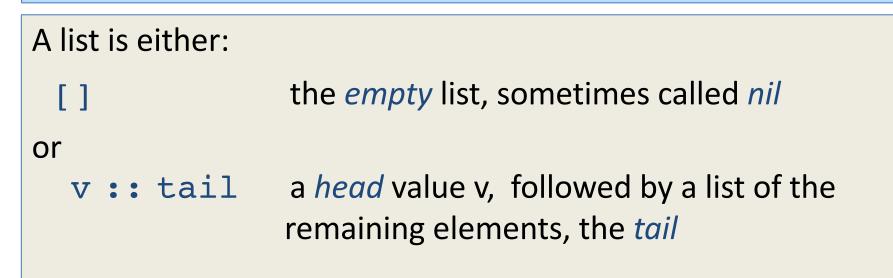
Programming Languages and Techniques (CIS120)

Lecture 4

Lists Recursion and Tuples

#### Review: What is a list?



- 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

### 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:

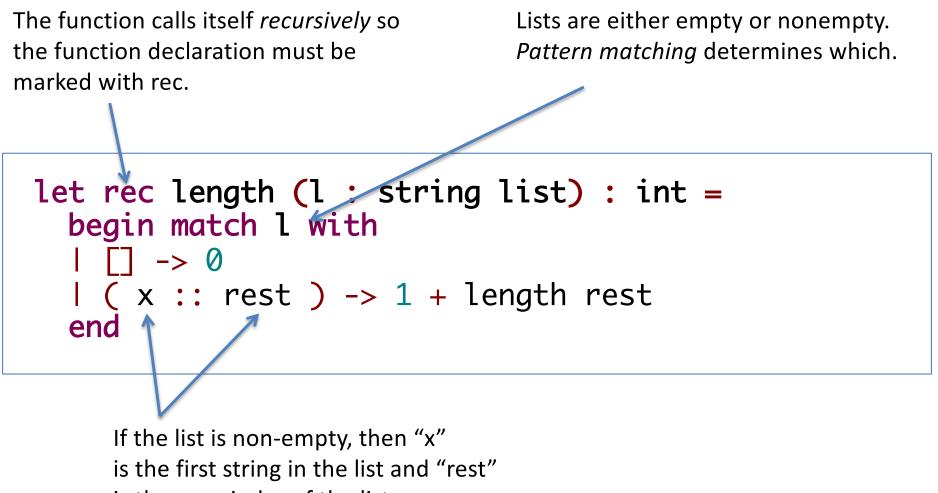
length []

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

- length (1::2::3::[]) = 1 + (length (2::3::[]))

  - = 1 + (length [])
  - 0

#### **Recursion Over Lists**



## **Calculating with Recursion**

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

```
→ (second case matches with rest = "b"::[])
```

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

 $\mapsto$  (substitute the list for I in the function body)

```
\mapsto (second case matches again, with rest = [])
```

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

```
\mapsto (substitute [] for I in the function body)
```

```
\mapsto 1 + 1 + 0 \Rightarrow 2
```

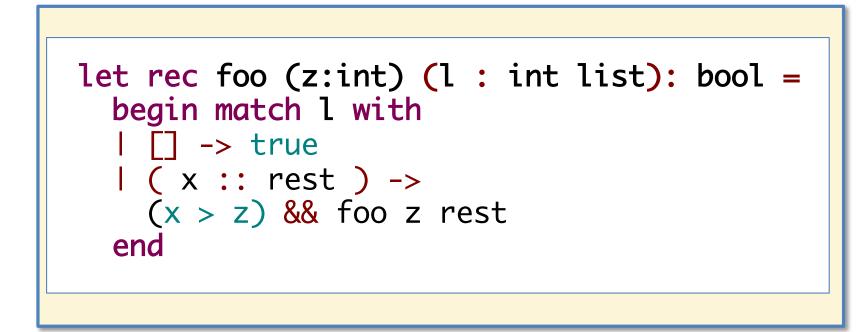
...

**CIS120** 

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

What best describes the behavior of the function call (foo 3 1)?

ANSWER: determines whether every element is greater than 3

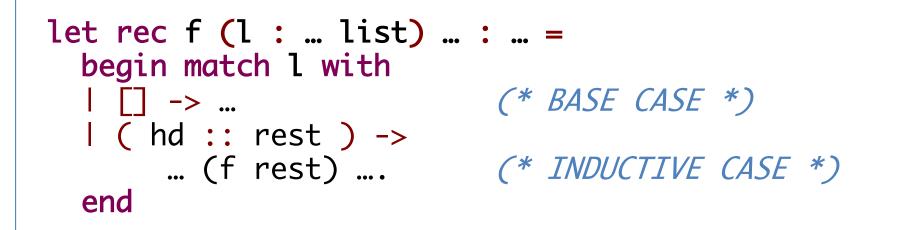


#### More examples...

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

#### **Structural Recursion Over Lists**

Structural recursion builds an answer from smaller components:



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

## Pattern Matching on Tuples

• Tuples can be inspected by pattern matching:

```
let first (x: string * int) : string =
    begin match x with
    l (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:

#### **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:

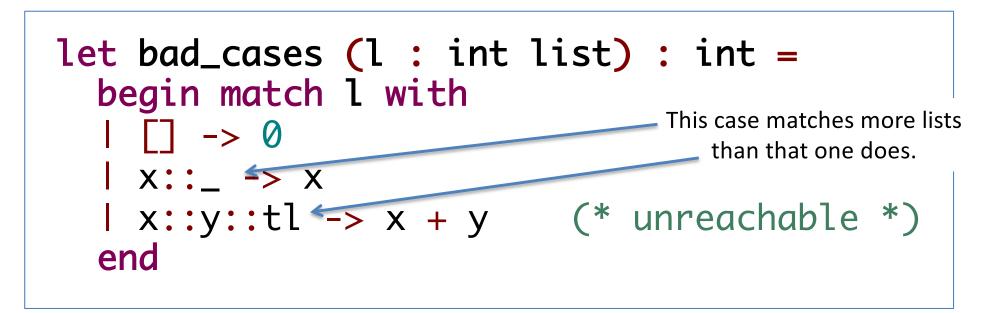
 x :: []
 matches lists with 1 element
 [x]
 matches lists with 1 element
 x:: (y::tl)
 matches lists of length at least 2
 (x::xs, y::ys)

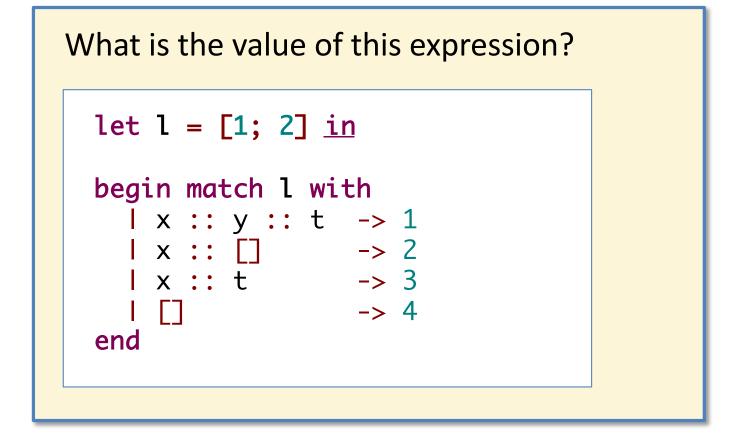
#### Wildcard Pattern

- 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 2<sup>nd</sup> part

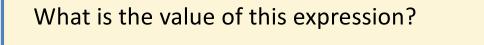
#### **Unused Branches**

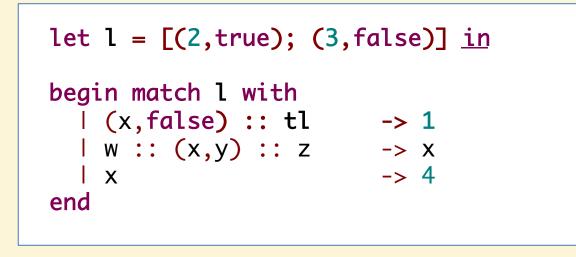
- The branches in a match expression are considered in order from top to bottom.
- If you have "redundant" matches, then some later branches might not be reachable.
  - OCaml will give you a warning in this case











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
    l 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
   l x::y::_ -> x+y
   l _ -> 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")]