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 Wednesday
A list is either:

\[ \text{[ ]} \]

the *empty* list, sometimes called *nil*

or

\[ v :: \text{tail} \]

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*.
Pattern Matching

OCaml provides a single expression called *pattern matching* for inspecting a list and naming its subcomponents.

```ocaml
let mylist : int list = [1; 2; 3; 5]
let y : int = begin
  match mylist with
  | []              -> 42
  | first :: rest    -> first + 10
end
```

Case analysis is justified because there are only two shapes a list can have.

Note that `first` and `rest` are identifiers that are bound in the body of the branch

- `first` names the head of the list; its type is the element type.
- `rest` names the tail of the list; its type is the list type

The type of the match expression is the (one) type shared by its branches.
Calculating with Matches

• Consider how to evaluate a match expression:

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

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 \(\text{first}::\text{rest}\) when \(\text{first}\) is 1 and \(\text{rest}\) is \((2::(3::[])))\).

So, substitute 1 for \(\text{first}\) in the second branch.

\[\rightarrow 1 + 10\]

\[\rightarrow 11\]
The Inductive Nature of Lists

A list is either:

- \[ \] the empty list, sometimes called \textit{nil}

or

- \( v :: \text{tail} \) a \textit{head} value \( v \), followed by a \textit{list} of the remaining elements, the \textit{tail}

• What is going on!? The definition of list mentions ‘list’!

• Insight: ‘list’ is \textit{inductive}:
  – The empty list \([\] \) is the (only) list of 0 elements
  – To construct a list of \((1+n)\) elements, add an element to an \textit{existing} list of \( n \) elements
  – The set of list values contains \textit{all and only} values constructed this way

• Corresponding computation principle: \textit{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:

\[
\text{length (1::2::3::[])} = 1 + (\text{length (2::3::[])}), \\
\text{length (2::3::[])} = 1 + (\text{length (3::[])}), \\
\text{length (3::[])} = 1 + (\text{length []}), \\
\text{length []} = 0.
\]
The function calls itself *recursively* so the function declaration must be marked with rec.

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

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

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"]

$\Rightarrow (\text{substitute the list for } l \text{ in the function body})$

\[
\begin{align*}
\text{begin match } \text{"a"::"b"::[]} \text{ with} \\
\mid [] & \Rightarrow 0 \\
\mid (x :: \text{rest} ) & \Rightarrow 1 + \text{length rest}
\end{align*}
\]

$\Rightarrow (\text{second case matches with rest } = \text{"b"::[]})$

\[
1 + (\text{length } \text{"b"::[]})
\]

$\Rightarrow (\text{substitute the list for } l \text{ in the function body})$

\[
1 + (\text{begin match } \text{"b"::}[] \text{ with} \\
\mid [] & \Rightarrow 0 \\
\mid (x :: \text{rest} ) & \Rightarrow 1 + \text{length rest}
\text{end})
\]

$\Rightarrow (\text{second case matches again, with rest } = [])$

\[
1 + (1 + \text{length } [])
\]

$\Rightarrow (\text{substitute } [] \text{ for } l \text{ in the function body})$

\[
\ldots
\]

\[
1 + 1 + 0 \Rightarrow 2
\]
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:

```ocaml
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:

```ocaml
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 ->
Mixing Tuples and Lists

- Tuples and lists can mix freely:

```c
[(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
• 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\(^{nd}\) 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?

```
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?

```haskell
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?

```haskell
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:

```ocaml
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:

```ocaml
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
Recursive function patterns

Recursive functions over lists follow a general pattern:

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

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

Structural recursion builds an answer from smaller components:

```ocaml
let rec f (l : ... list) ... : ... =
  begin match l with
  | [] -> ...
  | (hd :: rest) -> ... f rest ...
  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
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?
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
```