

Programming Languages and Techniques (CIS120)

Lecture 4

September 2, 2015

Lists and Tuples

Announcements

- Please bring your clickers to class every day
 - Clicker-based attendance starts *friday*
- Read Chapters 3 and 4 of the lecture notes
- HW#1 due Tuesday at midnight
- HW#2 will be available Friday
 - Due: following Tuesday: 9/15

Interactive Interlude

email.ml

Tuples and Tuple Patterns

Forms of Structured Data

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

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

Tuples

- In OCaml, tuples are created by writing the values, separated by commas, in parentheses:

```
let my_pair = (3, true)
let my_triple = ("Hello", 5, false)
let my_quaduple = (1,2,"three",false)
```

- Tuple types are written using '*'
 - e.g. `my_triple` has type:

```
string * int * bool
```

Pattern Matching 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, the pattern follows the syntax of the corresponding values
 - The patterns simply name the subcomponents

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)
```


What is the type of this expression?

```
(1, [1], [[1]])
```

1. int
2. int list
3. int list list
4. (int * int list) list
5. int * (int list) * (int list list)
6. (int * int * int) list
7. *none (expression is ill typed)*

Answer: 5

What is the type of this expression?

```
[ (1,true); (0, false) ]
```

1. `int * bool`
2. `int list * bool list`
3. `(int * bool) list`
4. `(int * bool) list list`
5. *none (expression is ill typed)*

Answer: 3

What is the type of this expression?

```
(1 :: [], 2 :: [], 3 :: [])
```

1. int
2. int list
3. int list list
4. int list * int list * int list
5. int * int list * int list list
6. (int * int * int) list
7. *none (expression is ill typed)*

Answer: 4

Nested Patterns

- So far, we've seen simple patterns:

`[]` *matches empty list*

`x::tl` *matches nonempty list*

`(a,b)` *matches pairs*

`(a,b,c)` *matches triples*

- Like expressions, patterns can *nest*:

`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 pattern is the wildcard pattern: `_`
`_ :: t1` *matches a non-empty list, but only names tail*
`(_, x)` *matches a pair, but only names the 2nd part*
- A wildcard pattern indicates that the value of the corresponding subcomponent is irrelevant.
 - And hence needs no name.

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
```

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

1

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

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

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

This case matches more lists than that one does.

(* unreachable *)

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  
    | _ -> failwith "l must have >= 2 elts"  
  end
```

- OCaml will give you a warning and show an example of what isn't covered by your cases
- The wildcard pattern and failwith are useful tools for ensuring match coverage

More List Programming

see `lists.ml`