CSE 120/130

Introduction to

Programming Languages and Techniques

Fall 2000

Handout 3

Pattern Matching and Sorting

Patterns

Recall that `hd` and `tl` together "undo" the cons `::` operation. It often happens that we want simultaneously to extract, and to give names to, the head and tail of a list. Suppose, for example, we wanted to "rotate" a list `l`

```
# let l = [1; 2; 3] ;;
val l : int list = [1; 2; 3]

# (tl l) @ (hd l)
- : int list = [2; 3; 1]
```

We could also write this as

```
# match l with h :: t  -> t @ [h] ;;
...
- : int list = [2; 3; 1]
```

Here the `...` stands for a "grumble" that OCaml makes about your code. We'll ignore it for the time being and try to understand what is going on.

match ... with ... -> ....

```
# match l with h :: t  -> t @ [h] ;;
```

On the right-hand side of the arrow `->` we see the "cons" operator, but here it is used as a *pattern* which introduces the variables `x` and `y`, and binds them respectively to the head and tail of `l`. To see why this is a sensible notation look at the list `[1; 2; 3]`. We know this is another way of writing `1 :: (2 :: ( 3 :: []))`. So we can understand the `match` expression as

```
match 1 :: (2 :: ( 3 :: [])) with h :: t  -> t @ [h] ;;
```

Now if we look at what is each side of the `with`, we see that the two sides have the same form, and if we "bind" `h` to `1` and `t` to `(2 :: ( 3 :: []))` (i.e., `[2; 3]`) the two sides are the same.

Let us look more closely at the "grumble". Here is a function that uses `match` to rotate a list.

```
# let rotate l =
      match l with h :: t -> t @ [h]  ;;

Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val rotate : 'a list -> 'a list = <fun>
```

What has happened is that OCaml has observed that not every list matches the pattern `h :: t`. The empty list doesn't match this pattern.

The solution is to provide an alternative pattern:

```
# let rotate l =
      match l with
        [] -> []
      | h :: t -> t @ [h]  ;;
val rotate : 'a list -> 'a list = <fun>
```

The alternative branch of the pattern match is signalled by the vertical bar `|`. Note that this replaces the conditional that we would have otherwise used to write the function using `hd` and `tl`.

Here is another example of a simple function that can be rewritten using pattern matching.

```
# let rec repeat l =
    if l = [] then []
    else hd l :: hd l :: repeat (tl l);;

# repeat [1; 2; 3];;
- : int list = [1; 1; 2; 2; 3; 3]

# let rec repeat l =
    match l with
      [] -> []
    | h::t -> h :: h :: repeat t;;
```

In the second implementation we avoid the repeated use of `hd l`.

Examples of Pattern Matching

```
let rec reverse l =
    match l with
      [] -> []
    |  h :: t -> reverse t @ [h]

let rec sum l =
    match l with
      [] -> 0
    | h :: t -> h + sum t
```

What are the types of these functions?


```
let rec count l =
    match l with
      [] -> 0
    | _ :: t -> 1 + count t
```

This illustrates the use of the "wild card" variable `_` . In this case we don't need a name for the head of `l` so we just put a place holder which matches anything.

```
let rec last l =
    match l with
      [x] -> x
    | y -> last (tl y)
```

In this example we first try to match `l` with the list containing one element `x`. We could also have written this pattern `x :: []` .

If the match fails we match `l` to a variable `y` (so that `y=l`).

Incomplete matches

We could also have written the `last` function

```
# let rec last l =
    match l with
      [x] -> x
    | _ :: t -> last t
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val last : 'a list -> 'a = <fun>
```

The "grumble" is quite useful. It has told us that the patterns covers all possible values of `l`, and that there are some inputs on which the function may fail.

The function will work on lists that have at least one element. What happens on empty lists?

```
# last [];;
Uncaught exception: Match_failure...
```


Probably the best way to write this function is with a three-way match

```
let rec last l =
  match l with
    [] -> (* generate informative error *)
  | [x] -> x
  | _ :: t -> last t;;
```

We'll learn how to generate an "informative error" later.

More complicated examples of pattern matching

Here is a function to test whether a list of integers is sorted.

```
let rec sorted l =
  match  l with
    [] -> true
  | [_] -> true
  | h1 :: h2 :: t -> h1 <= h2 && sorted (h2 :: t)
```

The first line of the match expression says the empty list is sorted. The second line matches lists of length 1. It says that these are also sorted. The third line matches successive heads of the argument when it has two or more elements.

We could equivalently have written the second line of the match as:

```
  | _ :: [] = true
```


The order of pattern matching

Here is another implementation of the `null` predicate:

```
let null l =
  match l with
    [] -> true
  | _  -> false
```

The second pattern matches any argument (including `nil`). OCaml matches the patterns in the order in which they occur in the function definition.

What would have happened if we had written the following function?

```
let null l =
  match l with
    _  -> false
  | [] -> true
```

Other patterns

Patterns do not have to be list patterns – they can be quite general. For example the following is a factorial function:

```
let rec fact n =
    match n with
      0 -> 1
    | n -> n * fact(n-1)
```

Again, note that the order of patterns in this function is important.

Tuples

OCaml allows us to form *heterogeneous* tuples with the notation $x_1, x_2, \ldots, x_n$. Examples:

```
# 1, 2, 3 ;;
- : int * int * int = 1, 2, 3

# 2, "cat";;
- : int * string = 2, "cat"

# 1, ("cat",true), "dog";;
- : int * (string * bool) * string = 1, ("cat", true), "dog"
```

Unlike lists, tuples cannot be “extended” by adjoining new elements.

Decomposing tuples

Tuples can be taken apart with pattern matching.

```
# let p = (2, "bats") ;;
val p : int * string = 2, "bats"

# match p with (x, _) -> x;;
- : int = 2

# match p with (_ , y) -> y;;
- : string = "bats"
```

2-tuples are often called “pairs”.

More complex patterns

The basic forms of patterns that we have seen can be mixed together to achieve more complex effects. The following definitions mix list and tuple patterns:

```
let rec zip l =
  match l with
    l1, [] -> []
  | [], l2 -> []
  | h1::t1, h2::t2 -> (h1,h2) :: zip (t1,t2)
val zip : 'a list * 'b list -> ('a * 'b) list = <fun>

# zip ([1; 2; 3], [4; 5; 6]);;
- : (int * int) list = [1, 4; 2, 5; 3, 6]
```

```
let rec unzip l =
  match l with
    [] -> [], []
  | (h1,h2)::t ->
      match unzip t with  l1,l2 -> h1::l1, h2::l2
val unzip : ('a * 'b) list -> 'a list * 'b list = <fun>

# unzip [1,2; 3,4; 5,6; 9,8];;
- : int list * int list = [1; 3; 5; 9], [2; 4; 6; 8]
```

The function `unzip` illustrates a number of points we have covered in pattern matching. It is a good idea to make sure you understand it.

Why use pattern matching?

It looks as though pattern matching isn't buying us much and in some cases is making our programs *longer*. Why use it?

For many of the programs we have written, it hasn't helped a great deal, but when we write more involved programs it may make them more readable.

Also there is an issue of *efficiency*. Consider the two values of `last`

```
let rec last l =
  match l with
    [x] -> x
  | y -> last (tl y);;
```

```
let rec last l =
    if null (tl l) then hd l
    else last(tl l)
```

In the second version we computed `(tl l)` twice. Using pattern matching is one way of avoiding this.

Sorting lists

Efficient sorting is a major concern in many parts of computer science. In particular, sorting large files is a major problem for commercial data processing, and sorting lists is similar in many respects to sorting files.

To sort a list we can use what should now be a familiar strategy: thinking recursively. Suppose we could sort the tail of a list. To get the whole list sorted, we'd need to insert the head in its correct position in a list that is already sorted.


```
(* insert(x,l) assumes that l is already sorted and inserts x
   into l so that the result is still sorted *)
let rec insert x l =
  match l with
    [] -> [x]
  | h::t -> if x <= h then x :: h :: t
            else h :: insert x t

val insert : 'a -> 'a list -> 'a list = <fun>

# insert 4 [1; 2; 5; 6];;
- : int list = [1; 2; 4; 5; 6]
```

To complete the sorting algorithm, we need only apply `insert` in turn to each element of the original list:

```
let rec sort l =
  match l with
    [] -> []
  | h::t -> insert h (sort t)

val sort : 'a list -> 'a list = <fun>

# sort [1; 9; 2; 8; 3; 7; 1; 6];;
- : int list = [1; 1; 2; 3; 6; 7; 8; 9]
```


Efficiency considerations

The sorting method we have just described is called *insertion sort*.

Suppose we sort a list that is in decreasing order. How much work is done by `sort`? We shall measure the work as the number of calls to `insert`, including both "top-level" calls made by `sort` and recursive calls made by `insert`.

Calling `insert`(x, l) on a list l of length m when x is placed at the end of the list requires m calls to `insert`.

Therefore calling `sort` on a list of length n requires

$$n + (n-1) + (n-2) + \ldots 1 = n(n-1)/2$$

calls to `insert`.

How long would sorting 100,000 records take with this method?

Quicksort

Quicksort is a much more efficient method of sorting. The idea is simply this:

- choose some element `x` ("at random") from the list;
- divide the list into three sub-lists: those elements less that `x`, those elements equal to `x`, and those elements greater than `x`;
- recursively sort the first and third of these; and
- concatenate the three lists together.

Here are two of the functions that give us the three sublists

```
let rec lessLst x l =
  match l with
    [] -> []
  | h::t -> if h < x then h :: lessLst x t
                else lessLst x t


let rec eqLst x l =
  match l with
    [] -> []
  | h::t -> if h = x then h :: eqLst x t
                else eqLst x t
```

It should be a "no brainer" to define the third.

Here is the code for quicksort. Notice that we take as our "random" choice, the head of the list.

```
let rec quickSort l =
  match l with
    [] -> []
  | [x] -> [x]
  | h::t -> quickSort (lessLst h t)
            @ h::eqLst h t
            @ quickSort (greaterLst h t)
```

Under what circumstances would the choice of the head as the "random" element be a very bad idea?

Quicksort – refining the code

Calling our three functions that split up a list causes three traversals of that list. On ML lists this is not a serious penalty, but on files stored on external media it is a problem – one would read the file three times over!

What we need is a function that splits the list into three in one traversal. Here we want our function to return more than one list, so we pack them together in a 3-tuple as shown in the following example

```
# split3 5 [1; 9; 2; 7; 3; 8; 2; 5; 6];;
- : int list * int list * int list = [1; 2; 3; 2], [5], [9; 7; 8; 6]
```

Here is the code for `split3`. We use `match ... with (s,e,g) -> ...` to "inspect" a 3-tuple.

```
let rec split3  x l =
  match l with
    [] -> [], [], []
  | h::t -> match split3 x t with (s, e, g) ->
              if h < x then (h::s, e,g)
              else if h = x then s, h::e, g
                   else s, e, h::g

val split3 : 'a -> 'a list -> 'a list * 'a list * 'a list = <fun>
```


Our code for quicksort is now

```
let rec quickSort l =
  match l with
    [] -> []
  | [x] -> [x]
  | h::t ->
      match split3 h t with (l, e, g) ->
        quickSort l @ h::e @ quickSort g
```

Assuming that the initial list is in random order, it can be shown that the *expected* running time for quicksort is proportional to $n \log_2 n$. How long would it now take to sort 100,000 records?

Mergesort

Suppose we have two lists that are already sorted and in ascending order. Here is a program `merge` that combines them into one list in ascending order. This function is useful in its own right and variations on it are widely used in data processing.

```
let rec merge p =
   match p with
     l1, [] -> l1
   | [], l2 -> l2
   | (h1::t1), (h2::t2) ->
       if h1 <= h2 then h1 :: merge (t1, h2::t2)
       else h2::merge (h1::t1, t2)

val merge : 'a list * 'a list -> 'a list = <fun>
```


An example of an application of `mergesort`:

```
# merge (["cat"; "dog"], ["bat"; "cow"; "emu"]);;
- : string list = ["bat"; "cat"; "cow"; "dog"; "emu"]
```

If `l1` has length n_1 and `l2` has length n_2, what is the maximum number of comparisons performed by `merge(l1,l2)`?

The next function we need is a function to split a list into two components. It returns a pair.

```
let rec split l =
  match l with
    [] -> [], []
  | [x] -> [x], []
  | h1 :: h2 :: t ->
      match split t with (l1, l2) -> h1::l1, h2 ::l2

val split : 'a list -> 'a list * 'a list = <fun>

# split [1; 2; 3; 4; 5; 6];;
- : int list * int list = [1; 3; 5], [2; 4; 6]
```


Now we can write the sorting function `mergesort`. The idea is simple: split the list into two, recursively mergesort the two halves, and merge the results.

```
let rec mergesort l =
   match l with
     [] -> []
   | [x] -> [x]
   | l -> match split l with (l1, l2) ->
            merge(mergesort l1, mergesort l2)

# mergesort [1; 2; 3; 4; 6; 0; 9; 7; 8; 5; 6; 4];;
- : int list = [0; 1; 2; 3; 4; 4; 5; 6; 6; 7; 8; 9]
```

The number of comparisons made by mergesort on a list of length n is, roughly speaking, proportional to $n \log_2 n$.

Packaging mergesort

```
let rec mergesort comp l =
  match l with
    [] -> []
  | [x] -> [x]
  | l ->
    let rec merge p =
       match p with
         l1, [] -> l1
       | [], l2 -> l2
       | (h1::t1),  (h2::t2) ->
           if comp h1 h2 then h1 :: merge(t1, h2::t2)
           else h2::merge(h1::t1, t2) in
```

(continued on next page)


```
    let rec split l =
      match l with
        [] -> [], []
      | [x] -> [x], []
      | h1 :: h2 :: t ->
          match split t with (l1, l2) ->  h1::l1, h2 ::l2  in

    match split l with (l1, l2) ->
       merge(mergesort comp l1, mergesort comp l2)

val mergesort : ('a -> 'a -> bool) -> 'a list -> 'a list = <fun>
```

Not only have we “encapsulated” mergesort here, we have generalized it to work for an arbitrary comparison function `comp`.

In this example, several uses of made of the name `l`. It would have been kinder to the reader to use different variable names. Be sure you understand which uses are the same.

Examples of using mergesort

```
# mergesort ( <= ) [1; 9; 3; 8; 4; 7; 3] ;;
- : int list = [1; 3; 3; 4; 7; 8; 9]

# mergesort ( >= ) [1; 9; 3; 8; 4; 7; 3] ;;
- : int list = [9; 8; 7; 4; 3; 3; 1]

# mergesort ( <= ) ["Jack"; "havoc"; "Newt"; "Alan"; "cat"];;
- : string list = ["Alan"; "Jack"; "Newt"; "cat"; "havoc"]

# let strComp s1 s2 = String.uppercase s1 <= String.uppercase s2 in
    mergesort strComp ["Jack"; "havoc"; "Newt"; "Alan"; "cat"]   ;;
- : string list = ["Alan"; "cat"; "havoc"; "Jack"; "Newt"]
```


Bottom-up mergesort

An alternative version of mergesort – which is more closely related to how mergesort is actually performed in secondary storage – is the following.

- Split the list into a list of single element lists. Each of these is trivially sorted.
- Repeatedly sweep through this list of lists merging successive pairs.
- When the list of lists has length 1, extract the single (sorted) list from it.

Bottom-up mergesort – a hand-worked trace

```
# let ll = singlist [1; 2; 3; 4; 6; 0; 9; 7; 8; 5; 6; 4];;
val ll : int list list =
  [[1]; [2]; [3]; [4]; [6]; [0]; [9]; [7]; [8]; [5]; [6]; [4]]

# let ll = sweep ll;;
val ll : int list list =
  [[1; 2]; [3; 4]; [0; 6]; [7; 9]; [5; 8]; [4; 6]]

# let ll = sweep ll;;
val ll : int list list = [[1; 2; 3; 4]; [0; 6; 7; 9]; [4; 5; 6; 8]]

# let ll = sweep ll;;
val ll : int list list = [[0; 1; 2; 3; 4; 6; 7; 9]; [4; 5; 6; 8]]

# let ll = sweep ll;;
val ll : int list list = [[0; 1; 2; 3; 4; 4; 5; 6; 6; 7; 8; 9]]
```