4 Lists and Patterns

This chapter will focus on two common elements of programming in OCaml: lists and pattern matching. Both of these were discussed in Chapter 2 (A Guided Tour), but we'll go into more depth here, presenting the two topics together and using one to help illustrate the other.

4.1 List Basics

An OCaml list is an immutable, finite sequence of elements of the same type. As we've seen, OCaml lists can be generated using a bracket-and-semicolon notation:

```
# open Base;;
# [1;2;3];;
- : int list = [1; 2; 3]
```

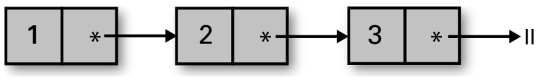
And they can also be generated using the equivalent `::` notation:

```
# 1 :: (2 :: (3 :: []));;
- : int list = [1; 2; 3]
# 1 :: 2 :: 3 :: [];;
- : int list = [1; 2; 3]
```

As you can see, the `::` operator is right-associative, which means that we can build up lists without parentheses. The empty list `[]` is used to terminate a list. Note that the empty list is polymorphic, meaning it can be used with elements of any type, as you can see here:

```
# let empty = [];;
val empty : 'a list = []
# 3 :: empty;;
- : int list = [3]
# "three" :: empty;;
- : string list = ["three"]
```

The way in which the `::` operator attaches elements to the front of a list reflects the fact that OCaml's lists are in fact singly linked lists. The figure below is a rough graphical representation of how the list `1 :: 2 :: 3 :: []` is laid out as a data structure. The final arrow (from the box containing `3`) points to the empty list.



Each :: essentially adds a new block to the preceding picture. Such a block contains two things: a reference to the data in that list element, and a reference to the remainder of the list. This is why :: can extend a list without modifying it; extension allocates a new list element but does not change any of the existing ones, as you can see:

```
# let l = 1 :: 2 :: 3 :: [];;
val l : int list = [1; 2; 3]
# let m = 0 :: l;;
val m : int list = [0; 1; 2; 3]
# l;;
- : int list = [1; 2; 3]
```

4.2 Using Patterns to Extract Data from a List

We can read data out of a list using a `match` expression. Here's a simple example of a recursive function that computes the sum of all elements of a list:

```
# let rec sum l =
    match l with
    | [] -> 0
    | hd :: tl -> hd + sum tl;;
val sum : int list -> int = <fun>
# sum [1;2;3];;
- : int = 6
# sum [];;
- : int = 0
```

This code follows the convention of using `hd` to represent the first element (or head) of the list, and `tl` to represent the remainder (or tail).

The `match` expression in `sum` is really doing two things: first, it's acting as a case-analysis tool, breaking down the possibilities into a pattern-indexed list of cases. Second, it lets you name substructures within the data structure being matched. In this case, the variables `hd` and `tl` are bound by the pattern that defines the second case of the match expression. Variables that are bound in this way can be used in the expression to the right of the arrow for the pattern in question.

The fact that `match` expressions can be used to bind new variables can be a source of confusion. To see how, imagine we wanted to write a function that filtered out from a list all elements equal to a particular value. You might be tempted to write that code as follows, but when you do, the compiler will immediately warn you that something is wrong:

```
# let rec drop_value l to_drop =
    match l with
    | [] -> []
    | to_drop :: tl -> drop_value tl to_drop
    | hd :: tl -> hd :: drop_value tl to_drop;;
```

```
Line 5, characters 7-15:
Warning 11 [redundant-case]: this match case is unused.
val drop_value : 'a list -> 'a -> 'a list = <fun>
```

Moreover, the function clearly does the wrong thing, filtering out all elements of the list rather than just those equal to the provided value, as you can see here:

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

So, what's going on?

The key observation is that the appearance of `to_drop` in the second case doesn't imply a check that the first element is equal to the value `to_drop` that was passed in as an argument to `drop_value`. Instead, it just causes a new variable `to_drop` to be bound to whatever happens to be in the first element of the list, shadowing the earlier definition of `to_drop`. The third case is unused because it is essentially the same pattern as we had in the second case.

A better way to write this code is not to use pattern matching for determining whether the first element is equal to `to_drop`, but to instead use an ordinary `if` expression:

```
# let rec drop_value l to_drop =
    match l with
    | [] -> []
    | hd :: tl ->
      let new_tl = drop_value tl to_drop in
      if hd = to_drop then new_tl else hd :: new_tl;;
val drop_value : int list -> int -> int list = <fun>
# drop_value [1;2;3] 2;;
- : int list = [1; 3]
```

If we wanted to drop a particular literal value, rather than a value that was passed in, we could do this using something like our original implementation of `drop_value`:

```
# let rec drop_zero l =
    match l with
    | [] -> []
    | 0  :: tl -> drop_zero tl
    | hd :: tl -> hd :: drop_zero tl;;
val drop_zero : int list -> int list = <fun>
# drop_zero [1;2;0;3];;
- : int list = [1; 2; 3]
```

4.3 Limitations (and Blessings) of Pattern Matching

The preceding example highlights an important fact about patterns, which is that they can't be used to express arbitrary conditions. Patterns can characterize the layout of a data structure and can even include literals, as in the `drop_zero` example, but that's where they stop. A pattern can check if a list has two elements, but it can't check if the first two elements are equal to each other.

You can think of patterns as a specialized sublanguage that can express a limited (though still quite rich) set of conditions. The fact that the pattern language is limited turns out to be a good thing, making it possible to build better support for patterns in the compiler. In particular, both the efficiency of `match` expressions and the ability of the compiler to detect errors in matches depend on the constrained nature of patterns.

4.3.1 Performance

Naively, you might think that it would be necessary to check each case in a `match` in sequence to figure out which one fires. If the cases of a match were guarded by arbitrary code, that would be the case. But OCaml is often able to generate machine code that jumps directly to the matched case based on an efficiently chosen set of runtime checks.

As an example, consider the following rather silly functions for incrementing an integer by one. The first is implemented with a `match` expression, and the second with a sequence of `if` expressions:

```
# let plus_one_match x =
    match x with
    | 0 -> 1
    | 1 -> 2
    | 2 -> 3
    | 3 -> 4
    | 4 -> 5
    | 5 -> 6
    | _ -> x + 1;;
val plus_one_match : int -> int = <fun>
# let plus_one_if x =
    if      x = 0 then 1
    else if x = 1 then 2
    else if x = 2 then 3
    else if x = 3 then 4
    else if x = 4 then 5
    else if x = 5 then 6
    else x + 1;;
val plus_one_if : int -> int = <fun>
```

Note the use of _ in the above match. This is a wildcard pattern that matches any value, but without binding a variable name to the value in question.

If you benchmark these functions, you'll see that `plus_one_if` is considerably slower than `plus_one_match`, and the advantage gets larger as the number of cases increases. Here, we'll benchmark these functions using the `core_bench` library, which can be installed by running `opam install core_bench` from the command line.

```
# #require "core_bench";;
# open Core_bench;;
# [ Bench.Test.create ~name:"plus_one_match" (fun () ->
        plus_one_match 10)
  ; Bench.Test.create ~name:"plus_one_if" (fun () ->
        plus_one_if 10) ]
  |> Bench.bench;;
Estimated testing time 20s (2 benchmarks x 10s). Change using -quota
    SECS.

 Name             Time/Run
 plus_one_match   34.86ns
 plus_one_if      54.89ns

- : unit = ()
```

Here's another, less artificial example. We can rewrite the `sum` function we described earlier in the chapter using an `if` expression rather than a match. We can then use the functions `is_empty`, `hd_exn`, and `tl_exn` from the `List` module to deconstruct the list, allowing us to implement the entire function without pattern matching:

```
# let rec sum_if l =
    if List.is_empty l then 0
    else List.hd_exn l + sum_if (List.tl_exn l);;
val sum_if : int list -> int = <fun>
```

Again, we can benchmark these to see the difference:

```
# let numbers = List.range 0 1000 in
  [ Bench.Test.create ~name:"sum_if" (fun () -> sum_if numbers)
  ; Bench.Test.create ~name:"sum"    (fun () -> sum numbers) ]
  |> Bench.bench;;
Estimated testing time 20s (2 benchmarks x 10s). Change using -quota
    SECS.

 Name     Time/Run

 sum_if   62.00us
 sum      17.99us

- : unit = ()
```

In this case, the `match`-based implementation is many times faster than the `if`-based implementation. The difference comes because we need to effectively do the same work multiple times, since each function we call has to reexamine the first element of the list to determine whether or not it's the empty cell. With a `match` expression, this work happens exactly once per list element.

This is a more general phenomenon: pattern matching is very efficient, and is usually faster than what you might write yourself.

4.3.2 Detecting Errors

The error-detecting capabilities of `match` expressions are if anything more important than their performance. We've already seen one example of OCaml's ability to find problems in a pattern match: in our broken implementation of `drop_value`, OCaml warned us that the final case was redundant. There are no algorithms for determining if a predicate written in a general-purpose language is redundant, but it can be solved reliably in the context of patterns.

OCaml also checks `match` expressions for exhaustiveness. Consider what happens if we modify `drop_zero` by deleting the handler for one of the cases. As you can see, the compiler will produce a warning that we've missed a case, along with an example of an unmatched pattern:

```
# let rec drop_zero l =
    match l with
    | [] -> []
    | 0  :: tl -> drop_zero tl;;
Lines 2-4, characters 5-31:
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
1::_
val drop_zero : int list -> 'a list = <fun>
```

Even for simple examples like this, exhaustiveness checks are pretty useful. But as we'll see in Chapter 7 (Variants), they become yet more valuable as you get to more complicated examples, especially those involving user-defined types. In addition to catching outright errors, they act as a sort of refactoring tool, guiding you to the locations where you need to adapt your code to deal with changing types.

4.4 Using the List Module Effectively

We've so far written a fair amount of list-munging code using pattern matching and recursive functions. In real life, you're usually better off using the `List` module, which is full of reusable functions that abstract out common patterns for computing with lists.

Let's work through a concrete example. We'll write a function `render_table` that, given a list of column headers and a list of rows, prints them out in a well-formatted text table. When we're done, here's how the resulting function should work:

```
# Stdio.print_endline
    (render_table
       ["language";"architect";"first release"]
       [ ["Lisp" ;"John McCarthy" ;"1958"] ;
         ["C"    ;"Dennis Ritchie";"1969"] ;
         ["ML"   ;"Robin Milner"  ;"1973"] ;
         ["OCaml";"Xavier Leroy"  ;"1996"] ;
  ]);;
| language | architect      | first release |
|----------+----------------+---------------|
| Lisp     | John McCarthy  | 1958          |
| C        | Dennis Ritchie | 1969          |
| ML       | Robin Milner   | 1973          |
| OCaml    | Xavier Leroy   | 1996          |
- : unit = ()
```

The first step is to write a function to compute the maximum width of each column of data. We can do this by converting the header and each row into a list of integer lengths, and then taking the element-wise max of those lists of lengths. Writing the code for all of this directly would be a bit of a chore, but we can do it quite concisely by making use of three functions from the `List` module: `map`, `map2_exn`, and `fold`.

`List.map` is the simplest to explain. It takes a list and a function for transforming elements of that list, and returns a new list with the transformed elements. Thus, we can write:

```
# List.map ~f:String.length ["Hello"; "World!"];;
```

```
- : int list = [5; 6]
```

List.map2_exn is similar to List.map, except that it takes two lists and a function for combining them. Thus, we might write:

```
# List.map2_exn ~f:Int.max [1;2;3] [3;2;1];;
- : int list = [3; 2; 3]
```

The _exn is there because the function throws an exception if the lists are of mismatched length:

```
# List.map2_exn ~f:Int.max [1;2;3] [3;2;1;0];;
Exception: (Invalid_argument "length mismatch in map2_exn: 3 <> 4")
```

List.fold is the most complicated of the three, taking three arguments: a list to process, an initial accumulator value, and a function for updating the accumulator. List.fold walks over the list from left to right, updating the accumulator at each step and returning the final value of the accumulator when it's done. You can see some of this by looking at the type-signature for fold:

```
# List.fold;;
- : 'a list -> init:'accum -> f:('accum -> 'a -> 'accum) -> 'accum = <fun>
```

We can use List.fold for something as simple as summing up a list:

```
# List.fold ~init:0 ~f:(+) [1;2;3;4];;
- : int = 10
```

This example is particularly simple because the accumulator and the list elements are of the same type. But fold is not limited to such cases. We can for example use fold to reverse a list, in which case the accumulator is itself a list:

```
# List.fold ~init:[] ~f:(fun acc hd -> hd :: acc) [1;2;3;4];;
- : int list = [4; 3; 2; 1]
```

Let's bring our three functions together to compute the maximum column widths:

```
# let max_widths header rows =
    let lengths l = List.map ~f:String.length l in
    List.fold rows
      ~init:(lengths header)
      ~f:(fun acc row ->
          List.map2_exn ~f:Int.max acc (lengths row));;
val max_widths : string list -> string list list -> int list = <fun>
```

Using List.map we define the function lengths, which converts a list of strings to a list of integer lengths. List.fold is then used to iterate over the rows, using map2_exn to take the max of the accumulator with the lengths of the strings in each row of the table, with the accumulator initialized to the lengths of the header row.

Now that we know how to compute column widths, we can write the code to generate the line that separates the header from the rest of the text table. We'll do this in part by mapping String.make over the lengths of the columns to generate a string of dashes of the appropriate length. We'll then join these sequences of dashes together using String.concat, which concatenates a list of strings with an optional separator string, and ^, which is a pairwise string concatenation function, to add the delimiters on the outside:

```
# let render_separator widths =
    let pieces = List.map widths
        ~f:(fun w -> String.make w '-')
    in
    "|-" ^ String.concat ~sep:"-+-" pieces ^ "-|";;
val render_separator : int list -> string = <fun>
# render_separator [3;6;2];;
- : string = "|-----+--------+----|"
```

Note that we make the line of dashes two larger than the provided width to provide some whitespace around each entry in the table.

Performance of `String.concat` and `^`

In the preceding code we've concatenated strings two different ways: `String.concat`, which operates on lists of strings; and `^`, which is a pairwise operator. You should avoid `^` for joining large numbers of strings, since it allocates a new string every time it runs. Thus, the following code

```
# let s = "." ^ "."  ^ "."  ^ "."  ^ "."  ^ "."  ^ ".";;
val s : string = "......."
```

will allocate strings of length 2, 3, 4, 5, 6 and 7, whereas this code

```
# let s = String.concat [".";".";".";".";".";".";"."];;
val s : string = "......."
```

allocates one string of size 7, as well as a list of length 7. At these small sizes, the differences don't amount to much, but for assembling large strings, it can be a serious performance issue.

Now we need code for rendering a row with data in it. We'll first write a function called `pad`, for padding out a string to a specified length:

```
# let pad s length =
    s ^ String.make (length - String.length s) ' ';;
val pad : string -> int -> string = <fun>
# pad "hello" 10;;
- : string = "hello     "
```

We can render a row of data by merging together the padded strings. Again, we'll use `List.map2_exn` for combining the list of data in the row with the list of widths:

```
# let render_row row widths =
    let padded = List.map2_exn row widths ~f:pad in
    "| " ^ String.concat ~sep:" | " padded ^ " |";;
val render_row : string list -> int list -> string = <fun>
# render_row ["Hello";"World"] [10;15];;
- : string = "| Hello      | World           |"
```

Finally, we can bring this all together to build the `render_table` function we wanted at the start!

```
# let render_table header rows =
    let widths = max_widths header rows in
    String.concat ~sep:"\n"
```

```
       (render_row header widths
        :: render_separator widths
        :: List.map rows ~f:(fun row -> render_row row widths)
       );;
val render_table : string list -> string list list -> string = <fun>
```

4.4.1 More Useful List Functions

The previous example touched on only three of the functions in `List`. We won't cover the entire interface (for that you should look at the online docs[1]), but a few more functions are useful enough to mention here.

Combining List Elements with `List.reduce`

`List.fold`, which we described earlier, is a very general and powerful function. Sometimes, however, you want something simpler and easier to use. One such function is `List.reduce`, which is essentially a specialized version of `List.fold` that doesn't require an explicit starting value, and whose accumulator has to consume and produce values of the same type as the elements of the list it applies to.

Here's the type signature:

```
# List.reduce;;
- : 'a list -> f:('a -> 'a -> 'a) -> 'a option = <fun>
```

`reduce` returns an optional result, returning `None` when the input list is empty.

Now we can see `reduce` in action:

```
# List.reduce ~f:(+) [1;2;3;4;5];;
- : int option = Some 15
# List.reduce ~f:(+) [];;
- : int option = None
```

Filtering with `List.filter` and `List.filter_map`

Very often when processing lists, you want to restrict your attention to a subset of the values on your list. The `List.filter` function is one way of doing that:

```
# List.filter ~f:(fun x -> x % 2 = 0) [1;2;3;4;5];;
- : int list = [2; 4]
```

Sometimes, you want to both transform and filter as part of the same computation. In that case, `List.filter_map` is what you need. The function passed to `List.filter_map` returns an optional value, and `List.filter_map` drops all elements for which `None` is returned.

Here's an example. The following function computes a list of file extensions from a list of files, piping the results through `List.dedup_and_sort` to return the list with duplicates removed and in sorted order. Note that this example uses `String.rsplit2` from the String module to split a string on the rightmost appearance of a given character:

[1] https://v3.ocaml.org/p/base/v0.14.3/doc/Base/List/index.html

```
# let extensions filenames =
    List.filter_map filenames ~f:(fun fname ->
        match String.rsplit2 ~on:'.' fname with
        | None  | Some ("",_) -> None
        | Some (_,ext) ->
          Some ext)
    |> List.dedup_and_sort ~compare:String.compare;;
val extensions : string list -> string list = <fun>
# extensions ["foo.c"; "foo.ml"; "bar.ml"; "bar.mli"];;
- : string list = ["c"; "ml"; "mli"]
```

The preceding code is also an example of an or-pattern, which allows you to have multiple subpatterns within a larger pattern. In this case, `None | Some ("",_)` is an or-pattern. As we'll see later, or-patterns can be nested anywhere within larger patterns.

Partitioning with `List.partition_tf`

Another useful operation that's closely related to filtering is partitioning. The function `List.partition_tf` takes a list and a function for computing a Boolean condition on the list elements, and returns two lists. The `tf` in the name is a mnemonic to remind the user that `true` elements go to the first list and `false` ones go to the second. Here's an example:

```
# let is_ocaml_source s =
    match String.rsplit2 s ~on:'.' with
    | Some (_,("ml"|"mli")) -> true
    | _ -> false;;
val is_ocaml_source : string -> bool = <fun>
# let (ml_files,other_files) =
  List.partition_tf ["foo.c"; "foo.ml"; "bar.ml"; "bar.mli"]
    ~f:is_ocaml_source;;
val ml_files : string list = ["foo.ml"; "bar.ml"; "bar.mli"]
val other_files : string list = ["foo.c"]
```

Combining lists

Another very common operation on lists is concatenation. The `List` module actually comes with a few different ways of doing this. There's `List.append`, for concatenating a pair of lists.

```
# List.append [1;2;3] [4;5;6];;
- : int list = [1; 2; 3; 4; 5; 6]
```

There's also `@`, an operator equivalent of `List.append`.

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

In addition, there is `List.concat`, for concatenating a list of lists:

```
# List.concat [[1;2];[3;4;5];[6];[]];;
- : int list = [1; 2; 3; 4; 5; 6]
```

Here's an example of using `List.concat` along with `List.map` to compute a recursive listing of a directory tree.

```
# module Sys = Core.Sys
  module Filename = Core.Filename;;
module Sys = Core.Sys
module Filename = Core.Filename
# let rec ls_rec s =
    if Sys.is_file_exn ~follow_symlinks:true s
    then [s]
    else
      Sys.ls_dir s
      |> List.map ~f:(fun sub -> ls_rec (Filename.concat s sub))
      |> List.concat;;
val ls_rec : string -> string list = <fun>
```

Note that this example uses some functions from the `Sys` and `Filename` modules from `Core` for accessing the filesystem and dealing with filenames.

The preceding combination of `List.map` and `List.concat` is common enough that there is a function `List.concat_map` that combines these into one, more efficient operation:

```
# let rec ls_rec s =
    if Sys.is_file_exn ~follow_symlinks:true s
    then [s]
    else
      Sys.ls_dir s
      |> List.concat_map ~f:(fun sub -> ls_rec (Filename.concat s
    sub));;
val ls_rec : string -> string list = <fun>
```

4.5 Tail Recursion

The only way to compute the length of an OCaml list is to walk the list from beginning to end. As a result, computing the length of a list takes time linear in the size of the list. Here's a simple function for doing so:

```
# let rec length = function
    | [] -> 0
    | _ :: tl -> 1 + length tl;;
val length : 'a list -> int = <fun>
# length [1;2;3];;
- : int = 3
```

This looks simple enough, but you'll discover that this implementation runs into problems on very large lists, as we'll show in the following code:

```
# let make_list n = List.init n ~f:(fun x -> x);;
val make_list : int -> int list = <fun>
# length (make_list 10);;
- : int = 10
# length (make_list 10_000_000);;
Stack overflow during evaluation (looping recursion?).
```

The preceding example creates lists using `List.init`, which takes an integer `n` and a function `f` and creates a list of length `n`, where the data for each element is created by calling `f` on the index of that element.

To understand where the error in the above example comes from, you need to learn a bit more about how function calls work. Typically, a function call needs some space to keep track of information associated with the call, such as the arguments passed to the function, or the location of the code that needs to start executing when the function call is complete. To allow for nested function calls, this information is typically organized in a stack, where a new *stack frame* is allocated for each nested function call, and then deallocated when the function call is complete.

And that's the problem with our call to `length`: it tried to allocate 10 million stack frames, which exhausted the available stack space. Happily, there's a way around this problem. Consider the following alternative implementation:

```
# let rec length_plus_n l n =
    match l with
    | [] -> n
    | _ :: tl -> length_plus_n tl (n + 1);;
val length_plus_n : 'a list -> int -> int = <fun>
# let length l = length_plus_n l 0;;
val length : 'a list -> int = <fun>
# length [1;2;3;4];;
- : int = 4
```

This implementation depends on a helper function, `length_plus_n`, that computes the length of a given list plus a given `n`. In practice, `n` acts as an accumulator in which the answer is built up, step by step. As a result, we can do the additions along the way rather than doing them as we unwind the nested sequence of function calls, as we did in our first implementation of `length`.

The advantage of this approach is that the recursive call in `length_plus_n` is a *tail call*. We'll explain more precisely what it means to be a tail call shortly, but the reason it's important is that tail calls don't require the allocation of a new stack frame, due to what is called the *tail-call optimization*. A recursive function is said to be *tail recursive* if all of its recursive calls are tail calls. `length_plus_n` is indeed tail recursive, and as a result, `length` can take a long list as input without blowing the stack:

```
# length (make_list 10_000_000);;
- : int = 10000000
```

So when is a call a tail call? Let's think about the situation where one function (the *caller*) invokes another (the *callee*). The invocation is considered a tail call when the caller doesn't do anything with the value returned by the callee except to return it. The tail-call optimization makes sense because, when a caller makes a tail call, the caller's stack frame need never be used again, and so you don't need to keep it around. Thus, instead of allocating a new stack frame for the callee, the compiler is free to reuse the caller's stack frame.

Tail recursion is important for more than just lists. Ordinary non-tail recursive calls are reasonable when dealing with data structures like binary trees, where the depth of the tree is logarithmic in the size of your data. But when dealing with situations where the depth of the sequence of nested calls is on the order of the size of your data, tail recursion is usually the right approach.

4.6 Terser and Faster Patterns

Now that we know more about how lists and patterns work, let's consider how we can improve on an example from Chapter 2.3.2 (Recursive List Functions): the function remove_sequential_duplicates. Here's the implementation that was described earlier:

```
# let rec remove_sequential_duplicates list =
    match list with
    | [] -> []
    | [x] -> [x]
    | first :: second :: tl ->
      if first = second then
        remove_sequential_duplicates (second :: tl)
      else
        first :: remove_sequential_duplicates (second :: tl);;
val remove_sequential_duplicates : int list -> int list = <fun>
```

We'll consider some ways of making this code more concise and more efficient.

First, let's consider efficiency. One problem with the above code is that it in some cases re-creates on the right-hand side of the arrow a value that already existed on the left-hand side. Thus, the pattern [hd] -> [hd] actually allocates a new list element, when really, it should be able to just return the list being matched. We can reduce allocation here by using an as pattern, which allows us to declare a name for the thing matched by a pattern or subpattern. While we're at it, we'll use the function keyword to eliminate the need for an explicit match:

```
# let rec remove_sequential_duplicates = function
    | [] as l -> l
    | [_] as l -> l
    | first :: (second :: _ as tl) ->
      if first = second then
        remove_sequential_duplicates tl
      else
        first :: remove_sequential_duplicates tl;;
val remove_sequential_duplicates : int list -> int list = <fun>
```

We can further collapse this by combining the first two cases into one, using an *or-pattern*:

```
# let rec remove_sequential_duplicates list =
    match list with
    | [] | [_] as l -> l
    | first :: (second :: _ as tl) ->
      if first = second then
        remove_sequential_duplicates tl
      else
        first :: remove_sequential_duplicates tl;;
val remove_sequential_duplicates : int list -> int list = <fun>
```

We can make the code slightly terser now by using a `when` clause. A `when` clause allows us to add an extra precondition to a pattern in the form of an arbitrary OCaml expression. In this case, we can use it to include the check on whether the first two elements are equal:

```
# let rec remove_sequential_duplicates list =
    match list with
    | [] | [_] as l -> l
    | first :: (second :: _ as tl) when first = second ->
      remove_sequential_duplicates tl
    | first :: tl -> first :: remove_sequential_duplicates tl;;
val remove_sequential_duplicates : int list -> int list = <fun>
```

Polymorphic Compare

You might have noticed that `remove_sequential_duplicates` is specialized to lists of integers. That's because `Base`'s default equality operator is specialized to integers, as you can see if you try to apply it to values of a different type.

```
# open Base;;
# "foo" = "bar";;
Line 1, characters 1-6:
Error: This expression has type string but an expression was expected
    of type
         int
```

OCaml also has a collection of polymorphic equality and comparison operators, which we can make available by opening the module `Base.Poly`.

```
# open Base.Poly;;
# "foo" = "bar";;
- : bool = false
# 3 = 4;;
- : bool = false
# [1;2;3] = [1;2;3];;
- : bool = true
```

Indeed, if we look at the type of the equality operator, we'll see that it is polymorphic.

```
# (=);;
- : 'a -> 'a -> bool = <fun>
```

If we rewrite `remove_sequential_duplicates` with `Base.Poly` open, we'll see that it gets a polymorphic type, and can now be used on inputs of different types.

```
# let rec remove_sequential_duplicates list =
    match list with
    | [] | [_] as l -> l
    | first :: (second :: _ as tl) when first = second ->
      remove_sequential_duplicates tl
    | first :: tl -> first :: remove_sequential_duplicates tl;;
val remove_sequential_duplicates : 'a list -> 'a list = <fun>
# remove_sequential_duplicates [1;2;2;3;4;3;3];;
- : int list = [1; 2; 3; 4; 3]
# remove_sequential_duplicates ["one";"two";"two";"two";"three"];;
- : string list = ["one"; "two"; "three"]
```

OCaml comes with a whole family of polymorphic comparison operators, including the standard infix comparators, <, >=, etc., as well as the function `compare` that returns `-1`, `0`, or `1` to flag whether the first operand is smaller than, equal to, or greater than the second, respectively.

You might wonder how you could build functions like these yourself if OCaml didn't come with them built in. It turns out that you *can't* build these functions on your own. OCaml's polymorphic comparison functions are built into the runtime to a low level. These comparisons are polymorphic on the basis of ignoring almost everything about the types of the values that are being compared, paying attention only to the structure of the values as they're laid out in memory. (You can learn more about this structure in Chapter 24 (Memory Representation of Values).)

Polymorphic compare does have some limitations. For example, it will fail at runtime if it encounters a function value.

```
# (fun x -> x + 1) = (fun x -> x + 1);;
Exception: (Invalid_argument "compare: functional value")
```

Similarly, it will fail on values that come from outside the OCaml heap, like values from C bindings. But it will work in a reasonable way for most other kinds of values.

For simple atomic types, polymorphic compare has the semantics you would expect: for floating-point numbers and integers, polymorphic compare corresponds to the expected numerical comparison functions. For strings, it's a lexicographic comparison.

That said, experienced OCaml developers typically avoid polymorphic comparison. That's surprising, given how obviously useful it is, but there's a good reason. While it's very convenient, in some cases, the type oblivious nature of polymorphic compare means that it does something that doesn't make sense for the particular type of values you're dealing with. This can lead to surprising and hard to resolve bugs in your code. It's for this reason that `Base` discourages the use of polymorphic compare by hiding it by default.

We'll discuss the downsides of polymorphic compare in more detail in Chapter 15 (Maps and Hash Tables).

Note that `when` clauses have some downsides. As we noted earlier, the static checks associated with pattern matches rely on the fact that patterns are restricted in what they can express. Once we add the ability to add an arbitrary condition to a pattern, something is lost. In particular, the ability of the compiler to determine if a match is exhaustive, or if some case is redundant, is compromised.

Consider the following function, which takes a list of optional values, and returns the number of those values that are `Some`. Because this implementation uses `when` clauses, the compiler can't tell that the code is exhaustive:

```
# let rec count_some list =
    match list with
    | [] -> 0
    | x :: tl when Option.is_none x -> count_some tl
    | x :: tl when Option.is_some x -> 1 + count_some tl;;
Lines 2-5, characters 5-57:
```

```
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
_::_
(However, some guarded clause may match this value.)
val count_some : 'a option list -> int = <fun>
```

Despite the warning, the function does work fine:

```
# count_some [Some 3; None; Some 4];;
- : int = 2
```

If we add another redundant case without a `when` clause, the compiler will stop complaining about exhaustiveness and won't produce a warning about the redundancy.

```
# let rec count_some list =
    match list with
    | [] -> 0
    | x :: tl when Option.is_none x -> count_some tl
    | x :: tl when Option.is_some x -> 1 + count_some tl
    | x :: tl -> -1 (* unreachable *);;
val count_some : 'a option list -> int = <fun>
```

Probably a better approach is to simply drop the second `when` clause:

```
# let rec count_some list =
    match list with
    | [] -> 0
    | x :: tl when Option.is_none x -> count_some tl
    | _ :: tl -> 1 + count_some tl;;
val count_some : 'a option list -> int = <fun>
```

This is a little less clear, however, than the direct pattern-matching solution, where the meaning of each pattern is clearer on its own:

```
# let rec count_some list =
    match list with
    | [] -> 0
    | None   :: tl -> count_some tl
    | Some _ :: tl -> 1 + count_some tl;;
val count_some : 'a option list -> int = <fun>
```

The takeaway from all of this is although `when` clauses can be useful, we should prefer patterns wherever they are sufficient.

As a side note, the above implementation of `count_some` is longer than necessary; even worse, it is not tail recursive. In real life, you would probably just use the `List.count` function:

```
# let count_some l = List.count ~f:Option.is_some l;;
val count_some : 'a option list -> int = <fun>
```