Fold Left vs. Fold Right

Having built both `fold_right` and `fold_left`, it's worthwhile to compare and contrast them. The immediately obvious difference is the order in which they combine elements from the list: right to left vs. left to right. When the operator being used to combine elements is *associative*, that order doesn't doesn't change the final value of the computation. But for non-associative operators like `(-)`, it can:

```
# List.fold_right (-) [1;2;3] 0;;  (* 1 - (2 - (3 - 0)) *)
- : int = 2
# List.fold_left (-) 0 [1;2;3];;   (* ((0 - 1) - 2) - 3 *)
- : int = -6
```

A second difference is that `fold_left` is tail recursive whereas `fold_right` is not. So if you need to use `fold_right` on a very lengthy list, you may instead want to reverse the list first then use `fold_left`; the operator will need to take its arguments in the reverse order, too:

```
# List.fold_right (fun x y -> x - y) [1;2;3] 0;;
- : int = 2

# List.fold_left (fun y x -> x - y) 0 (List.rev [1;2;3]);;
- : int = 2

# List.fold_left (fun x y -> y - x) 0 (List.rev (0--1_000_000));;
- : int = 500000

# List.fold_right (fun y x -> x - y) (0--1_000_000) 0;;
Stack overflow during evaluation (looping recursion?)
```

Of course we could have written `fun x y -> x - y` as `(-)` in the code above, but we didn't in this one case just so you could see how the argument order has to change. Recall that `(--)` is an operator we defined in the recitation on lists; `x--y` tail-recursively computes the list containing all the integers from `x` to `y` inclusive:

```
let (--) i j =
  let rec from i j l =
    if i>j then l
    else from i (j-1) (j::l)
    in from i j []
```

We could even define a tail-recursive version of `fold_right` by baking in the list reversal:

```
let fold_right_tr f l accu =
  List.fold_left (fun acc elt -> f elt acc) accu (List.rev l)

# fold_right_tr (fun x y -> x - y) (0--1_000_000) 0;;
- : int = 500000
```

A third difference is the types of the functions. It can be hard to remember what those types are! Luckily we can always ask the toplevel:

```
# List.fold_left;;
- : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>

# List.fold_right;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

To understand those types, look for the list argument in each one of them. That tells you the type of the values in the list. Then look for the type of the return value; that tells you the type of the accumulator. From there you can work out everything else.

- In `fold_left`, the list argument is of type `'b list`, so the list contains values of type `'b`. The return type is `'a`, so the accumulator has type `'a`. Knowing that, we can figure out that the second argument is the initial value of the accumulator (because it has type `'a`). And we can figure out that the first argument, the combining operator, takes as its own first argument an accumulator value (because it has type `'a`), as its own second argument a list element (because it has type `'b`), and returns a new accumulator value.
- In `fold_right`, the list argument is of type `'a list`, so the list contains values of type `'a`. The return type is `'b`, so the accumulator has type `'b`. Knowing that, we can figure out that the third argument is the initial value of the accumulator (because it has type `'b`). And we can figure out that the first argument, the combining operator, takes as its own second argument an accumulator value (because it has type `'b`), as its own first argument a list element (because it has type `'a`), and returns a new accumulator value.

You might wonder why the argument orders are different between the two `fold` functions. I have no good answer to that question. Other libraries do in fact use different argument orders.

If you find it hard to keep track of all these argument orders, the `ListLabels` module in the standard library can help. It uses labeled arguments to give names to the combining operator (which it calls `f`) and the initial accumulator value (which it calls `init`). Internally, the implementation is actually identical to the `List` module.

```
# ListLabels.fold_left;;
- : f:('a -> 'b -> 'a) -> init:'a -> 'b list -> 'a = <fun>

# ListLabels.fold_right;;
- : f:('a -> 'b -> 'b) -> 'a list -> init:'b -> 'b = <fun>

# ListLabels.fold_left ~f:(fun x y -> x - y) ~init:0 [1;2;3];;
- : int = -6

# ListLabels.fold_right ~f:(fun y x -> x - y) ~init:0 [1;2;3];;
- : int = -6
```

Notice how in the two applications of fold above, we are able to write the arguments in a uniform order thanks to their labels. However, we still have to be careful about which argument to the combining operator is the list element vs. the accumulator value.

A digression on labeled arguments and fold

It's possible to write our own version of the fold functions that would label the arguments to the combining operator, so we don't even have to remember their order:

```
let rec fold_left ~op:(f: acc:'a -> elt:'b -> 'a) ~init:accu l =
  match l with
    [] -> accu
  | a::l -> fold_left ~op:f ~init:(f ~acc:accu ~elt:a) l

let rec fold_right ~op:(f: elt:'a -> acc:'b -> 'b) l ~init:accu =
  match l with
    [] -> accu
  | a::l -> f ~elt:a ~acc:(fold_right ~op:f l ~init:accu)
```

But those functions aren't as useful as they might seem:

```
# fold_left ~op:(+) ~init:0 [1;2;3];;
Error: This expression has type int -> int -> int
but an expression was expected of type acc:'a -> elt:'b -> 'a
```

The problem is that the built-in `(+)` operator doesn't have labeled arguments, so we can't pass it in as the combining operator to our labeled functions. We'd have to define our own labeled version of it:

```
# let add ~acc ~elt = acc+elt;;
val add : acc:int -> elt:int -> int = <fun>

# fold_left ~op:add ~init:0 [1;2;3];;
- : int = 6
```

But now we have to remember that the `~acc` parameter to `add` will become the left-hand argument to `(+)`. That's not really much of an improvement over what we had to remember to begin with.

</i>