CS251 Programming Languages **Handout # 20**
Prof. Lyn Turbak **February 16, 2007**
Wellesley College

Higher-Order List Functions

One of the commandments of computer science is *thou shalt abstract over common patterns of computation*. Upon seeing that two code fragments share similar structure, a good programmer will write a function whose body captures the commonalities and whose parameters express what is different between the fragments. Then the two code fragments can be expressed as two invocations of the same function on different arguments. For example, suppose we see the following pattern of pair addition:

```
... let ((a,b),(c,d)) = (p,q) in let (p',q') = (a+c,b+d) in ...
... let ((w,x),(y,z)) = (r,s) in let (r',s') = (w+y,x+z) in ...
```

Then we should introduce a function that captures this pattern:

```
let add_pairs ((x1,y1),(x2,y2)) = (x1+x2,y1+y2)
... let (p',q') = add_pairs (p,q) in ...
... let (r',s') = add_pairs (r,s) in ...
```

First-class functions are essential tools for abstracting over common idioms. Often what differs between two similar patterns can only be expressed with function parameters. For example, to capture the pattern in

```
... let ((a,b),(c,d)) = (p,q) in let (p',q') = (a+c,b+d) in ...
... let ((w,x),(y,z)) = (r,s) in let (r',s') = (w-y,x-z) in ...
```

we need to abstract over the fact that `+` is used in the first case and `-` is used in the second. We can do this with a functional argument:

```
let glue_pairs f ((x1,y1),(x2,y2)) = (f x1 x2, f y1 y2)
... let (p',q') = glue_pairs (+) (p,q) in ...
... let (r',s') = glue_pairs (-) (r,s) in ...
```

In this handout we explore how first class functions enable abstracting over common list processing idioms. We will use these abstractions heavily throughout the rest of the semester.

1 List Transformation: Mapping

Consider the following `map_sq` function:

```
let rec map_sq xs =
  match xs with
    [] -> []
  | x::xs' -> (x*x)::(map_sq xs')
```

If we want to instead increment each element of the list rather than square it, we would write the function `map_inc`:

```
let rec map_inc xs =
  match xs with
    [] -> []
  | x::xs' -> (x+1)::(map_inc xs')
```

The idiom of applying a function to each element of a list is so common that it deserves to be captured into a function, which is traditionally called `map`:

```
let rec map f xs =
  match xs with
    [] -> []
  | x::xs' -> (f x)::(map f xs')
```

Given `map`, it is easy to define `map_sq` and `map_inc`:

```
# let map_sq xs = map (fun x -> x*x) xs;;
val map_sq : int list -> int list = <fun>

# let map_inc ys = map (fun x -> x+1) ys;;
val map_inc : int list -> int list = <fun>
```

Interestingly, we can define `map_sq` and `map_inc` without naming the list arguments:

```
# let map_sq = map (fun x -> x*x);;
val map_sq : int list -> int list = <fun>

# let map_inc = map (fun x -> x+1);;
val map_inc : int list -> int list = <fun>
```

In these examples, we are partially applying the curried `map` function by supplying it only with its function argument. It returns a function that expects the second argument (the input list) and returns the resulting list. There is no need to name the input list. These simplifications are instances of a general simplification known as **eta-reduction**, which says that `fun x -> f x` can be simplified to `f` for any function `f`.

It's not necessary to name mappers. As show in Fig. 1, we can use `map` directly wherever we need it. These examples highlight that `map` can be used on any type of input and output lists. Indeed, the type of `map` inferred by OCAML is:

```
val map : ('a -> 'b) -> 'a list -> 'b list
```

So `map` uses an `'a -> 'b` function to map an `'a list` to a `'b list`.

```
# map ((-) 1) [6;4;3;5;8;7;1];;

# map ((flip (-)) 1) [6;4;3;5;8;7;1];;

# map (fun z -> (z mod 2) = 0) [6;4;3;5;8;7;1];;

# map (fun w -> (w, w*w)) [6;4;3;5;8;7;1];;

# map (fun ys -> 6::ys) [[7;2;4];[3];[];[1;5]];;

# map (glue_pairs (+)) [((1,2),(3,4)); ((8,5),(6,7))];;

# map app5 (map to_the [0;1;2;3;4]);;
```

Figure 1: Examples of `map`.

The examples also show how partially applied curried functions (such as `((-) 1)`, `((flip (-)) 1)`, and `(glue_pairs (+))`) can be used as functional arguments to `map`. This is a benefit of defining

multiple argument functions in curried form rather than tupled form. Sometimes we introduce new curried functions because they are useful in generating functional arguments to higher-order functions like `map`. For example, there is no prefix consing function in OCAML(`(::)` does *not* work), so we define

```
# let cons x xs = x :: xs;;
val cons : 'a -> 'a list -> 'a list = <fun>
```

Now we can write

```
# let mapcons x ys = map (cons x) ys;;
val mapcons : 'a -> 'a list list -> 'a list list = <fun>

# mapcons 6 [[7;2;4];[3];[];[1;5]];;
- : int list list = [[6; 7; 2; 4]; [6; 3]; [6]; [6; 1; 5]]
```

Programmers new to the notion of higher-order functions make some predictable mistakes when using higher order functions like `map`. Here's an *incorrect* attempt to define a function that doubles each integer in a list that is often seen from such programmers:

```
let map_dbl xs = map (x * 2) xs (* INCORRECT DECLARATION *)
```

There are two main things wrong with this definition:

1. The variable `x` is not declared anywhere and so is undefined. Perhaps there is a naive expectation that OCAML will understand that `x` is intended to range over the elements of `xs`, but it won't. Instead, OCAML will determine that `x` is a is a so-called **free variable** and will flag it as an error.

2. Even in the case where `x` happens to be declared earlier to be an integer that's available to this definition, the expression `(x * 2)` would have type `int`. But the first argument to `map` must have a type of the form `'a -> 'b` – i.e., it must be a *function*. In `map_dbl`, it should have type `int -> int`, not `int`.

Both problems can be fixed by introducing a function value using the `fun` syntax:

```
let map_dbl xs = map (fun x -> x * 2) xs (* CORRECT DECLARATION *)
```

The `fun x ->` ... introduces a parameter `x` that is bound in the body expression `x * 2`, so `x` is no longer a free variable. And `fun` creates a value with function type, which resolves the type problem.

For beginners, a good strategy is start by using `fun` explicitly in any situation that requires a functional argument or result. For example, it is *always* safe to invoke map using the template

```
map (fun x -> body) list
```

In this template, we think of `x` as being bound to each of the elements of *list* one by one to compute *body*. The answers are then collected into the resulting list. Once a definition is correct, it can sometimes be made more concise by using eta reduction and/or partial applications of curried functions to simplify the function parameter. For example, `fun x -> 2 * x` is equivalent to `fun x -> (( * ) 2) x`[1], which is equivalent to `(( * ) 2)`, and the function declaration `let map_dbl xs = map (( * ) 2) xs` is equivalent to `let map_dbl = map (( * ) 2)`.

Sometimes it's helpful to map over two lists at the same time. We accomplish this via `map2`:

[1]We write `( * )` rather than `(*)` because the later would be misinterpreted by OCAML as the beginning of a comment!

```
let rec map2 f xs ys =
  match (xs,ys) with
    ([], _) -> []
  | (_, []) -> []
  | (x::xs',y::ys') -> (f x y) :: map2 f xs' ys'
```

For example:

```
# map2 (+) [1;2;3] [40;50;60];;


# map2 (fun b x -> if b then x+1 else x*2) [true;false;false;true] [3;4;5;6];;


# let pair x y = (x,y);;
val pair : 'a -> 'b -> 'a * 'b = <fun>

# map2 pair [1;2;3;4] ['a';'b';'c'];;
```

As illustrated in the last example, `map2` ignores extra elements if one list is longer than the other. This is not the only way to handle lists with unequal length. OCaml provides a `List.map2` function that instead raises an exception if the list have unequal length. OCaml's `List.map` function is the same as the `map` defined above.

We can generalize the last example to the handy `zip` function:

```
# let zip (xs,ys) = map2 pair xs ys;;
val zip : 'a list * 'b list -> ('a * 'b) list = <fun>
```

2 List Transformation: Filtering

The `map` function is a **list transformer**: it takes a list as an input and returns another list as an output. Another list transformation is **filtering**, in which a given list is processed into another list that contains those elements from the input list that satisfy a given predicate (in the same relative order). While mapping produces an output list that has the same length as the input, filtering produces an output list whose length is less than or equal to the length of the input list.

As an example, consider the following `evens` procedure, which filters all the even elements from a given list:

```
let rec evens xs =
  match xs with
    [] -> []
  | x::xs' -> if (x mod 2) = 0 then x::(evens xs') else evens xs'
```

This is an instance of a more general filtering idiom, in which a predicate `p` determines which elements of the given list should be kept in the result:

```
# let rec filter p xs =
    match xs with
      [] -> []
    | x :: xs'-> if p x then x :: filter p xs' else filter p xs'
val filter : ('a -> bool) -> 'a list -> 'a list
```

For example:

```
# filter (fun x -> (x mod 2) = 0) [6;4;3;5;8;7;1];;
```

```
# filter ((flip (>)) 4) [6;4;3;5;8;7;1];;
```

```
# filter (fun x -> (abs (x - 4)) >= 2) [6;4;3;5;8;7;1];;
```

The `filter` function is available in the OCAML library as `List.filter`.

3 List Accumulation: Folding

3.1 `foldr` Encapsulates the Divide/Conquer/Glue Idiom on Lists

A common way to consume a list is to recursively accumulate a value from back to front starting at the base case and combining each element with the result of processing the rest of the elements. For example, here is an integer list summation function that uses this strategy:

```
# let rec sum xs =
    match xs with
      [] -> 0
    | (x::xs') -> x + sum xs'
val sum : int list -> int = <fun>
```

This pattern of list accumulation is captured by the `foldr` function:

```
# let rec foldr binop null xs =
    match xs with
      [] -> null
    | x :: xs' -> binop x (foldr binop null xs')
val foldr : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b
```

Given a list of elements $xs = x_1, x_2, \ldots, x_k$, a binary operator b, and a null value n, `foldr` b n xs yields the value $(b\ x_1\ (b\ x_2\ (\ldots\ (b\ x_k\ n)\ \ldots)))$. The name `foldr` comes from the fact that this function folds (combines) the elements of the list from **r**ight to left.

`foldr` is "the mother of all list recursive functions" because it captures the idiom of the divide/conquer/glue problem-solving strategy on lists. In the general case, `foldr` divides the list into head and tail (`x :: xs'`), conquers the tail by recursively processing it (`foldr binop null xs'`), and glues the head to the result for the tail via `binop`. It is also necessary to specify the result for the empty case (`null`). Because divide/conquer/glue is an incredibly effective strategy for processing lists, almost any list recursion can be expressed by supplying `foldr` with an appropriate `binop` and `null` (although in some cases we'll see that these can be rather complex).

In general, a function that expresses divide/conquer/glue over a recursive data structure is known as a **catamorphism**. `foldr` is the catamorphism for lists. Later in the course we shall encounter catamorphisms for other recursive data structures.

A strategy for defining a list recursive function *fcn* in terms of `foldr` is to begin with the template:

```
let fcn xs = foldr (fun x ans -> body) null xs
```

where *null* is the result of *`fcn`* `[]` and *body* needs to be fleshed out. For example to define a `sum` function that sums the elements of a list, we begin with

```
let sum xs = foldr (fun x ans -> body) 0 xs.
```

In `(fun x ans ->` *`body`*`)`, `x` stands for the current element being processed (the head of the

list) and `ans` stands for the result of recursively processing the tail of the list. For example, in `sums [7;3;6;4]`, the outermost `x` is `7` and the outermost `ans` is `3 + 6 + 4 = 13`. We want to combine these with `+` to yield `20`. So the fleshed out definition is:

```
let sum xs = foldr (fun x ans -> x + ans) 0 xs.
```

In this case, we can write the definition more succinctly as:

```
let sum xs = foldr (+) 0 xs.
```

Consider another example: the function `all_positive`, which returns `true` if all elements of a list are positive and `false` otherwise. Since `all_positive []` is `true` (each of the zero numbers in `[]` is positive), our template is:

```
let all_positive xs = foldr (fun x ans -> body) true xs.
```

In `(fun x ans -> body)`, `x` will stand for an element of the list (a number) and `ans` will stand for the result of processing the rest of the list (a boolean indicating if all the rest of the elements are positive). The appropriate body to combine `x` and `ans` in this case is `(x > 0) && ans`, yielding the definition:

```
let all_positive xs = foldr (fun x ans -> (x > 0) && ans) true xs.
```

Let's do one more example: the list reversal function `reverse`. Since `reverse []` is `[]`, our template is:

```
let reverse xs = foldr (fun x ans -> body) [] xs.
```

In our combining function, `x` will stand for an element of the list, and `ans` will stand for the result of reversing the rest of the elements in the list. For example, in processing `[1;2;3;4]`, `x` will be `1`, and `ans` will be `[4;3;2]`. How do we combine these to yield the desired result `[4;3;2;1]`? Via `[4;3;2]@[1]`. Generalizing this concrete example yields the final definition:

```
let reverse xs = foldr (fun x ans -> ans @ [x]) [] xs.
```

Figs. 2–3 show examples of using `foldr` to define a variety of other functions. Note how classical list processing functions like `append`, `flatten`, and `mapcons`, and even higher-order list functions like `map` and `filter`, can be defined in terms of `foldr`.

We can make many of the definitions in Figs. 2–3 even shorter by using eta reduction to remove the list argument. For example, rather than writing

```
let prod ns = foldr ( * ) 1 ns
```

we could instead write

```
let prod = foldr ( * ) 1
```

Unfortunately, in some cases eta reduction interacts badly with OCaml's type reconstruction. For example, consider this alternative definition of `flatten` from Fig. 3:

```
# let flatten' = foldr (@) [];;
val flatten' : '_a list list -> '_a list = <fun>
```

Note the presence of the type variable `'_a` in place of the usual type variable `'a`. This is a restricted type variable that can denote *exactly one type* in the rest of the program.[2] For instance, suppose we first use `flatten'` on an `int list list`:

```
# flatten' [[7;2;4];[3];[];[1;5]];;
- : int list = [7; 2; 4; 3; 1; 5]
```

[2]It is introduced by the OCaml type reconstruction algorithm to satisfy something called the **value restriction**, which restricts polymorphism for `let`-bound expressions that are not syntactic functions. The value restriction is a simple way of fixing a subtle problem in type soundness that can be introduced by such expressions.

```
# let prod ns = foldr
val prod : int list -> int = <fun>

# prod [6;4;3;5;8;1;7];;
- : int = 20160

# let minlist ns = foldr
val minlist : int list -> int = <fun>

# minlist [6;4;3;5;8;1;7];;
- : int = 1

# let maxlist ns = foldr
val maxlist : int list -> int = <fun>

# maxlist [6;4;3;5;8;1;7];;
- : int = 8

# let all_even ns = foldr
val all_even : int list -> bool = <fun>

# all_even [6;4;3;5;8;1;7];;
- : bool = false

# all_even [6;4;8];;
- : bool = true

# let exists_positive ns = foldr
val exists_positive : int list -> bool = <fun>

# exists_positive [-3;-1;2;-5];;
- : bool = true

# exists_positive [-3;-1;-2;-5];;
- : bool = false
```

Figure 2: `foldr` examples, part 1

```
# let append xs ys = foldr
val append : 'a list -> 'a list -> 'a list = <fun>

# append [7;2;4] [3;1;5];;
- : int list = [7; 2; 4; 3; 1; 5]

# let flatten xss = foldr
val flatten : 'a list list -> 'a list = <fun>

# flatten [[7;2;4];[3];[];[1;5]];;
- : int list = [7; 2; 4; 3; 1; 5]

# let mapcons x yss = foldr
val mapcons : 'a -> 'a list list -> 'a list list = <fun>

# mapcons 6 [[7;2;4];[3];[];[1;5]];;
- : int list list = [[6; 7; 2; 4]; [6; 3]; [6]; [6; 1; 5]]

# let subsets xs = foldr
val subsets : 'a list -> 'a list list = <fun>

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

# let map f xs = foldr
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>

# let filter p xs = foldr
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
```

Figure 3: `foldr` examples, part 2

Now `'_a` is bound to `int list` and `flatten'` can *only* be applied to lists of integer lists. Any other application is an error:

```
# flatten' [['a';'b';'c'];['d'];[];['e';'f']];;
Characters 11-14:
  flatten' [['a';'b';'c'];['d'];[];['e';'f']];;
             ^^^
This expression has type char but is here used with type int
```

In cases where eta reduction introduces restricted type variables, we can often improve type reconstruction by putting back in the extra argument:

```
# let flatten xss = foldr (@) [] xss;;
val flatten : 'a list list -> 'a list = <fun>
# flatten [[7;2;4];[3];[];[1;5]];;
- : int list = [7; 2; 4; 3; 1; 5]
# flatten [['a';'b';'c'];['d'];[];['e';'f']];;
- : char list = ['a'; 'b'; 'c'; 'd'; 'e'; 'f']
```

A `foldr`-like function is available in the OCAML `List` module via the name `List.fold_right`. However, it differs from `foldr` in the order in which it takes its arguments. As shown by the following type, it takes its arguments in the following order: (1) (curried) binary operator (2) list to be folded and (3) null value:

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

3.2 `for_all`, `exists`, and `some`

The `all/exists` examples in Fig. 2 suggest some higher-order list functions for determining if all or some elements in a list satsify a predicate. For example, the following `for_all` function determines if all elements of a list satsify a predicate `p`:

```
# let for_all p xs = foldr (&&) true (map p xs)
val for_all : ('a -> bool) -> 'a list -> bool = <fun>
# let all_positive = for_all
val all_positive : int list -> bool = <fun>
# all_positive [6;4;3;5;8;1;7];;
- : bool = true
# let all_even = for_all
val all_even : int list -> bool = <fun>
# all_even [6;4;3;5;8;1;7];;
- : bool = false
```

The following `exists` function determines if at least one element of a given list satisfies a predicate `p`:

```
# let exists p xs = foldr (||) false (map p xs)
val exists : ('a -> bool) -> 'a list -> bool = <fun>
# let exists_positive = exists ((flip (>)) 0);;
val exists_positive : int list -> bool = <fun>
# exists_positive [-3;-1;2;-5];;
- : bool = true
```

```
# let exists_even = exists (fun x -> (x mod 2) == 0);;
val exists_even : int list -> bool = <fun>

# exists_even [7;1;3;9;5];;
- : bool = false
```

Sometimes we want the first value from a list that satisfies a predicte. Since a list may not contain such a value, we need some way of expressing that there might not be any. The OCAML `'a option` type is used in situations like this. The `Some` constructor, with type `'a -> 'a option`, is used to inject a value into the `option` type, while the `None` constructor, with type `'a option`, is used to indicate that the `option` type has no value. Pattern matching is used to distinguish these cases. For example:

```
# map (fun x -> match x with
                  Some(v) -> v*v
                | None -> 0)
      [Some 3; None; Some 5; Some 2; None];;
- : int list = [9; 0; 25; 4; 0]
```

Using the `option` type, we can declare a higher-order function that returns `Some` of the first element of the list satisfying the predicate and `None` if there isn't one:

```
# let some p = foldr (fun x ans -> if p x then Some x else ans) None;;
val some : ('a -> bool) -> 'a list -> 'a option = <fun>

# some ((flip (>)) 0) [-5; -2; 4; -3; 1];;
- : int option = Some 4

# some ((flip (>)) 0) [-5; -2; -4; -3; -1];;
- : int option = None
```

Just because we *can* define a list processing function in terms of `foldr` doesn't mean that it's *a good idea* to do so. For example, the `for_all`, `exists`, and `some` functions given above aren't very efficient because they necessarily test the predicte on *all* elements of the list. For example, if we apply `exists_even` to a thousand element list whose first element is even, it will still check all other 999 elements to see if they're even! For these functions, it's better to hand-craft versions that perform the minimum number of predicate tests:

```
let rec for_all p xs =
  match xs with
    [] -> true
  | x::xs' -> (p x) && for_all p xs'

let rec exists p xs =
  match xs with
    [] -> false
  | x::xs' -> (p x) || exists p xs'

let rec some p xs =
  match xs with
    [] -> None
  | x::xs' -> if p x then Some x else some p xs'
```

The `List` module provides functions `List.for_all` and `List.exists` that are equivalent to the `for_all` and `exists` defined above.

3.3 More `foldr` Examples

Although almost any list processing function *can* be written in terms of `foldr`, it may take a fair bit of cleverness to do this, and sometimes definitions can be rather complex. We illustrate this in the context of a few more examples.

tails

Consider a `tails` function that returns a list of a given list and all of its successive tails:

```
# tails [1;2;3;4];;
- : int list list = [[1; 2; 3; 4]; [2; 3; 4]; [3; 4]; [4]; []]
# tails [];;
- : '_a list list = [[]]
```

To define `tails` in terms of `foldr`, we can fill in the following template:

```
let tails2 xs = foldr (fun x ans -> body) [[]] xs
```

The null value of `[[]]` is determined by the expected answer for `tails []`. The rest of the template is suggested by the structure of `foldr`. In `(fun x ans -> body)`, `x` will be bound to the head of the list and `ans` will be bound to the result of recursively processing the tail. For example, when this function is applied to the first element of `[1;2;3]`, `x` will be bound to `1`, and `ans` will be bound to `[[2; 3]; [3]; []]` (i.e., the result of processing `[2;3]`). How do we combine `1` with `[[2; 3]; [3]; []]` to produce `[[1; 2; 3]; [2; 3]; [3]; []]`? We need to create the list `[1;2;3]` and prepend it to `ans`. We can create `[1;2;3]` by prepending `1` onto the first element of `ans`. This leads to the following defintion:

```
let tails2 xs = foldr (fun x ans -> (x::List.hd ans)::ans) [[]] xs
```

isSorted

The `isSorted` function determines if a list of elements is sorted from least to greatest according to `<=`. E.g., `isSorted [1;3;4;7;9]` is `true` while `isSorted [1;3;7;4;9]` is `false`. Can we define `isSorted` using `foldr` and friends? Yup! But it's tricky, since the `foldr` needs to accumulate a *pair* of values: (1) the head of the sublist (so that we can compare it with the head of the list) and (2) a boolean indicating whether the sublist is sorted. In the base case, the empty list has no head, so we'll use the `None` value of the `option` type to indicate this and use a `Some` value for all other cases. We arrive at the following definition:

```
let isSorted xs = snd (foldr (fun x (opt,ans) ->
                                match opt with
                                  None -> (Some x, true)
                                | Some y -> (Some x, (x <= y) && ans))
                             (None, false)
                             xs)
```

There are other ways to define `isSorted`. For example, suppose that we zip the list together with its tail to give a list of pairs:

```
# zip([1;3;7;4;9], List.tl [1;3;7;4;9])
- : (int * int) list = [(1, 3); (3, 7); (7, 4); (4, 9)]
```

Then a non-empty list is sorted if and only the first element of each pair is `<=` to the second. Since we can't take the tail of an empty list, we need to handle that case specially. The resulting definition is:[3]

[3]The function `List.tl` extracts the tail of a list. Its companion `List.hd` extracts the head. Although it is always possible to extract the head and tail by pattern matching, these are sometimes convenient.

```
let isSorted xs =
  match xs with
    [] -> true
  | _ -> for_all (fun (a,b) -> (a <= b)) (zip (xs, List.tl xs))
```

We can replace `(fun (a,b) -> (a <= b))` by `uncurry (<=)` if we introduce the following definition:

```
# let uncurry f (a,b) = f a b;;
val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>
```

3.4 `foldr'`

Functions like `isSorted` are tricky to define with `foldr` because they need information in the tail of the list in addition the value being accumulated (a boolean in the case of `isSorted`). Above, we invented a complex mechanism for pairing the tail information with the accumulated value. But a simpler approach is to provide an alternative version of `foldr` that explicitly supplies the combining function with the tail of the list in addition to the head and the accumulated value. We call this function `foldr'`. Its combining function is a ternary operator rather than a binary operator:

```
let rec foldr' ternop null xs =
  match xs with
    [] -> null
  | x :: xs' -> ternop x xs' (foldr' ternop null xs')
val foldr' : ('a -> 'a list -> 'b -> 'b) -> 'b -> 'a list -> 'b
```

For example, here is a version of `isSorted` defined in terms of `foldr'`:

```
let isSorted2 xs = foldr' (fun x xs' ans -> ans && (xs' = [] || x < List.hd xs')) true xs
```

3.5 `foldr2` and Friends

As we saw with `map2`, it is sometimes helpful to have list processing functions that process the elements of two lists in lock step. The `foldr2` function is a general function for accumulating values over two lists processed in lock step:

```
let rec foldr2 ternop null xs ys=
  match (xs,ys) with
    ([], _) -> null
  | (_, []) -> null
  | (x :: xs', y::ys') -> ternop x  y (foldr2 ternop null xs' ys')
```

For example:

```
let zip (xs,ys) = foldr2 (fun x y ans -> (x,y)::ans) [] xs ys

let map2 f = foldr2 (fun x y ans -> (f x y)::ans) []

let for_all2 p = foldr2 (fun x y ans -> ((p x y) && ans)) true

let exists2 p = foldr2 (fun x y ans -> ((p x y) || ans)) false

let some2 p = foldr2 (fun x y ans -> if (p x y) then Some (x,y) else ans) None
```

3.6 foldl

There are situations where we want to accumulate the values in a list from left to right rather than from right to left. This is accomplished by `foldl`:

```
# let rec foldl ans binop xs =
    match xs with
      [] -> ans
    | x :: xs' -> foldl (binop ans x) binop xs'
val foldl : 'a -> ('a -> 'b -> 'a) -> 'b list -> 'a
```

For associative and commutative operators like `+` and `*`, `foldl` calculates the same final answer as a corresponding `foldr`, though the intermediate values may be different. But for other operators, it behaves differently. For example:

```
# foldl 0 (+) [1;2;3;4];;
- : int = 10

# foldl 0 (-) [1;2;3;4];;
- : int = -10

# let rev xs = foldl [] (flip cons) xs;; (* linear-time list reversal *)
val rev : 'a list -> 'a list = <fun>

# rev [1;2;3;4];;
- : int list = [4; 3; 2; 1]

# let digits2int ds = foldl 0 (fun ans x -> x+(10*ans)) ds;;
val digits2int : int list -> int = <fun>

# digits2int [3;1;2;5];;
- : int = 3125
```

4 List Generation

4.1 gen

In addition to transforming and consuming lists, there are useful abstractions for producing lists. A handy abstraction for list generation is the following `gen` function:

```
# let rec gen next isDone seed =
    if isDone seed then
      []
    else
      seed :: (gen next isDone (next seed))
val gen : ('a -> 'a) -> ('a -> bool) -> 'a -> 'a list
```

This function generates a sequence of values starting with an initial seed value, and uses the `next` function to generate the next value in the sequence from the current one. Generation continues until the `isDone` predicate is satisfied. At that point, all the elements in the sequence (except for the one satisfying the `isDone` predicate) are returned in a list.

Here are some sample uses of `gen`:

```
# let range lo hi = gen ((+) 1) ((<) hi) lo ;;
val range : int -> int -> int list = <fun>
```

```
# range 7 19;;

# gen ((flip (/)) 2) ((=) 0) 100;;

# gen List.tl ((=) []) [1;2;3;4;5];; (* List.tl takes the tail of a list *)
```

The `gen` function can be viewed as an iteration abstraction that lists together all the intermediate states of an iteration. The `next` function indicates how to get from the current state to the next state, and the `isDone` function indicates when the iteration is done. The following examples show how iterative factorial and Fibonacci computations can be expressed with `gen`:

```
# let fact_states n = gen (fun (n,a) -> (n-1,n*a)) (fun (n,a) -> n = 0) (n,1)
val fact_states : int -> (int * int) list = <fun>
# fact_states 5;;

# let fibsTo n = gen (fun (a,b) -> (b,a+b)) (fun (a,b) -> a > n) (0,1)
val fibsTo : int -> (int * int) list = <fun>
# fibsTo 13;;

# map fst (fibsTo 100);;
```

4.2 iterate

The following `iterate` function is similar to `gen` but only returns the final state of an iteration rather than a list of all states:

```
let rec iterate next isDone state =
  if isDone state then
    state
  else
    iterate next isDone (next state)
```

For example:

```
# let facti n = snd (iterate (fun (x,a) -> (x-1,x*a)) (fun (x,_) -> x = 0) (n,1))
val facti : int -> int = <fun>
# facti 5;;
- : int = 120
# let fibi n =
    match iterate (fun (i,a,b) -> (i+1,b,a+b)) (fun (i,_,_) -> i = n) (0,0,1) with
      (_,ans,_) -> ans;;
val fibi : int -> int = <fun>
# fibi 10;;
- : int = 55
```

4.3 ana

We can generalize `gen` into a more flexible function known as an **anamorphism**:

```
# let rec ana g seed =
    match g seed with
      None -> []
    | Some(h,seed') -> h:: ana g seed'
val ana : ('a -> ('b * 'a) option) -> 'a -> 'b list = <fun>
```

For example:

```
let map' f = ana (fun xs -> match xs with
                              [] -> None
                            | x::xs' -> Some(f x, xs'))

let gen' next isDone =
  ana (fun x -> if isDone x then None else Some(x, next x))

let fibsTo' n =
  ana (fun (a,b) -> if a > n then None else Some(a, (b, a+b))) (0,1)
```

In general, an anamorphism creates instances of a recursive datatype while a catamorphism accumulates results over instances of a recursive datatype.