CS251 Programming Languages **Handout # 14**
Prof. Lyn Turbak **February 11, 2004**
Wellesley College

First-Class Functions

NOTE: This draft is still missing some examples.

Data and procedures and the values they amass,
Higher-order functions to combine and mix and match,
Objects with their local state, the messages they pass,
A property, a package, a control point for a catch —
In the Lambda Order they are all first-class.
One Thing to name them all, One Thing to define them,
One Thing to place them in environments and bind them,
In the Lambda Order they are all first-class.

–Abstract for the *Revised*^{4} *Report on the Algorithmic Language Scheme*,
MIT Artificial Intelligence Lab Memo 848b, November 1991

1 Functions as First-Class Values

The key feature that sets the functional programming paradigm apart from other paradigms is its treatment of functions as first-class values. A value is said to be **first-class** if it can be:

1. named by a variable;
2. passed as an argument to a function;
3. returned as the result of a function;
4. stored in a data structure;
5. created in any context.

You can tell a lot about a programming language by what its first-class values are. For example, integers are first-class in almost every language. But compound structures like records and arrays do not always satisfy all four first-class properties. For example, early versions of FORTRAN did not allow arrays to be stored in other arrays. Early versions of PASCAL allowed records and arrays to be passed to a function as a value but not to be returned from a function as a result. Modern versions of C do not allow arrays to be passed as arguments or returned as results from functions, though they do allow *pointers* to arrays to be passed in this fashion. When combined with the fact that the lifetime of a local array ends when the procedure it was declared in is exited, this leads to numerous subtle bugs that plague C programmers.

In Pascal, functions and procedures satisfy the properties 2 and 5 but not the others. C functions (more precisely, C function pointers) satisfy properties 1 through 4, but do not satisfy property 5, since all functions must be declared at top level.

Functions in OCAML (as well as in SCHEME and HASKELL) satisfy all five first-class properties. Unlike C functions, they can be created anywhere in a program by a `fun` expression. This is a source of tremendous power; it is hard to overemphasize the importance of `fun` and first-class functions. Functions that take other functions as arguments or return them as results are known as **higher-order functions**. We will see many examples of the power of higher-order functions in this course.

1.1 The Substitution Model

As we have seen, a `fun` expression is just an OCAML notation for a function value.. For example, the `fun` expression

```
fun (a,b) -> (a + b) / 2
```

is pronounced "a function of two arguments that divides the sum of its two arguments by 2". Such an expression can be used in the operator position of a function call:

```
# (fun (a,b) -> (a + b) / 2) 3 7;;
- : int = 5
```

We can understand the meaning of a function application by the **substitution model**. In this model, a function application rewrites to a copy of the function body in which the formal parameters have been replaced by the actual argument values. For example:

```
# (fun (a,b) -> (a + b) / 2) 3 7;;
⇒ (3 + 7) / 2
⇒ 10 / 2
⇒ 5
```

1.2 Naming Functions

By the naming property of first-class function, we can attach a name to the averaging function using `let`:

```
# let avg = fun (a,b) -> (a + b) / 2;;
val avg : int * int -> int = <fun>

# avg (8,10);;
- : int = 9
```

Note that `let` does not create a function, it just names one. Rather, it is `fun` that creates the function. This fact is unfortunately obscured by the fact that OCAML supports syntactic sugar for function definition that hides the `fun`. That is, the above definition can also be written as:

```
# let avg (a,b) = (a + b) / 2;;
val avg : int * int -> int = <fun>
```

Even though the `fun` is not explicit in the sugared form of definition, it is important to remember that it is still there!

The fact that functions are values implies that the operator position of a function call can be an arbitrary expression. E.g. the expression

```
(if n = 0 then avg else fun (x,y) -> x + y) (3,7)
```

returns `5` if `n` evaluates to `0` and otherwise returns `10`.

1.3 Passing Functions as Arguments

Functions can be used as arguments to other functions. Consider the following expressions:

```
# let app_3_5 = fun p -> p (3,5);;
(* Top-level environment now contains binding app_3_5 ↦ (fun p -> p (3,5)) *)
```

```
# app_3_5 (fun (x,y) -> x + y);;
⇒ (fun p -> p (3,5)) (fun (x,y) -> x + y)
⇒ (fun (x,y) -> x + y) (3,5)
⇒ 3 + 5
⇒ 8

# app_3_5 (fun (x,y) -> x * y);;
⇒ (fun p -> p (3,5)) (fun (x,y) -> x * y)
⇒ (fun (x,y) -> x * y) (3,5)
⇒ 3 * 5
⇒ 15

# app_3_5 avg
⇒ (fun p -> p (3,5)) (fun (a,b) -> (a + b) / 2)
⇒ (fun (a,b) -> (a + b) / 2) (3,5)
⇒ (3 + 5) / 2
⇒ 8 / 2
⇒ 4

# app_3_5 (fun (a,b) -> a)
⇒ (fun p -> p (3,5)) (fun (a,b) -> a)
⇒ (fun (a,b) -> a) (3,5)
⇒ 3

# app_3_5 (fun (a,b) -> b)
⇒ (fun p -> p (3,5)) (fun (a,b) -> b)
⇒ (fun (a,b) -> b) (3,5)
⇒ 5
```

1.4 Returning Functions as Results

Functions can be returned as results from other functions. For example, suppose that `expt` is an exponentiation function — i.e., `expt(b,p)` returns the result of raising the base *base* to the power *p*.

```
# let to_the = fun p -> (fun b -> expt(b,p))
(* Top-level environment now contains binding ↦ fun p -> (fun b -> expt(b,p)) *)

# let sq = to_the 2
⇒ let sq = (fun p -> (fun b-> expt(b,p))) 2
⇒ let sq = (fun b -> expt(b,2))
(* Top-level environment now contains binding sq ↦ (fun b -> expt(b,2)) *)

# sq 5
⇒ (fun b -> expt(b,2)) 5
⇒ expt(5,2)
⇒ 25

# (to_the 3) 5
⇒ ((fun p -> (fun b -> expt(b,p))) 3) 5
⇒ (fun b -> expt(b,3)) 5
⇒ expt(5,3)
⇒ 125
```

Note that the function resulting from a call to `to_the` must somehow "remember" the value of power that `to_the` was called with. As shown above, this "memory" is completely explained by

the substitution model, in which `to_the` returns a specialized copy of `(fun b -> expt(b,p))` in which `p` is a particular integer.

Observe that the `to_the` function effectively takes two arguments (power `p` and base `b`), but rather than taking them in a single tuple, it takes them "one at a time". That is, `to_the` takes the power `p` and returns a function that takes the base `b` and returns the result of raising `b` to `p`. Functions that take their arguments one at a time in this fashion are know as **curried**[1] functions.

The type of `to_the` is `int -> (int -> int)`, which can also be written `int -> int -> int`, since `->` is treated as a right-associative operator. An uncurried (which we will also call "tupled") version of `to_the` would have type `int * int -> int`. In OCAML libraries like the `List` module, most multi-argument functions are written in a curried style rather than a tupled style. This is because the result of applying a curried function to one argument yields another function, and such a function is likely to be useful as an argument to a higher-order function. For example, if `app5` is the function `(fun f -> f 5)`, then the application `app5 (to_the 2)` makes sense but the application `app5 (expt 2)` does not. (We would have to write `app5 (fun b -> expt(b,2))`.)

As an example of using curried functions, consider a variant of `app_3_5` that expects a curried two-argument function as its argument:

```
# let app_3_5' = fun f -> f 3 5;;
val app_3_5' : (int -> int -> 'a) -> 'a = <fun>
```

For example:

```
# app_3_5' to_the;;
- : int = 125
```

OCAML's infix operators can be "converted" to curried prefix operators by wrapping them in parentheses. For example, `(+)` is a function of type `int -> int -> int`:

```
# (+) 1 2;;
- : int = 3
```

We can use these with `app_3_5`:

```
# app_3_5' (+);;
- : int = 8
# app_3_5' (-);;
- : int = -2
```

How would we multiply 3 by 5 (be careful – this is tricky!)

With OCAML's syntactic sugar, we can dispense with one or more explicit `fun`s in a curried function declaration. For example, all of the following are equivalent declarations of `to_the`:

```
# let to_the = fun p -> fun b -> expt(b,p);;
val to_the : int -> int -> int = <fun>

# let to_the p = fun b -> expt(b,p);;
val to_the : int -> int -> int = <fun>

# let to_the p b = expt(b,p);;
val to_the : int -> int -> int = <fun>
```

We can write functions that both take functions as arguments and return them as results. For

[1] named after the logician Haskell Curry, who is also remembered in the name of the programming language HASKELL.

example, the following `flip` function swaps the arguments of a curried two-argument function:

```
# let flip f a b = f b a
val flip : ('a -> 'b -> 'c) -> 'b -> 'a -> 'c = <fun>

# (flip (-)) 3 2;;
- : int = -1

# (flip to_the) 5 2;;
- : int = 25

# app_3_5' (flip (<));;
- : bool = false
```

We can use the "memory" of substitution to create functions like `app_3_5'` via the following function:[2]

```
# let church_pair = fun x -> fun y -> fun f -> f x y
```

For example, `church_pair 3 5` returns a function equivalent to `app_3_5'` and and `church_pair 17 32` returns a function equivalent to `fun f -> f 17 32`.

Because `church_pair` creates a function that remembers two values, it is effectively a pairing operator. That is, `church_pair x y` in many ways acts like the tuple `(x,y)`. For example, we can write functions `church_fst` and `church_snd` that extract the left and right elements of this functional "pair":

```
# let church_fst p = p (fun a b -> a);;
val fst : (('a -> 'b -> 'a) -> 'c) -> 'c = <fun>

# church_fst (church_pair 17 32);;
⇒ (fun p -> p (fun a b -> a)) ((fun x y -> fun f -> f x y) 17 32)
⇒ (fun p -> p (fun a b -> a)) (fun f -> f 17 32)
⇒ (fun f -> f 17 32) (fun a b -> a)
⇒ (fun a b -> a) 17 32
17
```

How would `church_snd` be defined? What is the value of `(church_pair 17 32) (+)`?

Since any data structure can be made out of pairs, it is not surprising that any data structure can be implemented in terms of functions. In fact, you should start thinking of functions as just another kind of data structure! This semester we will see many examples of how abstract data types can be elegantly represented by functions.

Note that functions with "memory" are very similar to methods in object-oriented languages. Indeed, later in the semester we will see how numerous aspects of the object-oriented programming paradigm can be modeled using first-class functions.

1.5 Storing Functions in Data Structures

Functions can be stored in data structures, like tuples and lists:

[2] We name the function `church_pair` because it is a functional encoding of pairs invented by Alonzo Church.

```
# let fun_tuple = (to_the, (+), app_3_5', church_pair);;
val fun_tuple :
  (int -> int -> int) * (int -> int -> int) * ((int -> int -> 'a) -> 'a) *
  ('b -> 'c -> ('b -> 'c -> 'd) -> 'd) = (<fun>, <fun>, <fun>, <fun>)

# match fun_tuple with (f,g,h,k) -> (h f, k 1 2 g);;
- : int * bool = (125, true)
```

1.6 Creating Functions

Finally, functions can be created in any context. In many programming languages, such as C, functions can only be defined at "top-level"; it is not possible to declare one function inside of another function. But as seen above in the `to_the` and `pair` examples, the ability to specialize a function to "remember" values in its body hinges crucially on the ability to have one `fun` nested inside another. In this pattern, applying the outer `fun` can cause values to be substituted into the body of the inner `fun`, allowing the resulting abstraction to "remember" the values of the parameters to the outer one.

2 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 section 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.

2.1 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`.

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),

```
# map ((-) 1) [6;4;3;5;8;7;1];;
- : int list = [-5; -3; -2; -4; -7; -6; 0]

# map ((flip (-)) 1) [6;4;3;5;8;7;1];;
- : int list = [5; 3; 2; 4; 7; 6; 0]

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

# map (fun w -> (w, w*w)) [6;4;3;5;8;7;1];;
- : (int * int) list = [(6, 36); (4, 16); (3, 9); (5, 25); (8, 64); (7, 49); (1, 1)]

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

# map (glue_pairs (+)) [((1,2),(3,4)); ((8,5),(6,7))];;
- : (int * int) list = [(4, 6); (14, 12)]

# map app5 (map to_the [0;1;2;3;4]);;
- : int list = [1; 5; 25; 125; 625]
```

Figure 1: Examples of `map`.

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 numbe 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 -> x * 2` is equivalent to `fun x -> (( * ) 2) x`[3], 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`:

```
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];;
- : int list = [41; 52; 63]

# map2 (fun b x -> if b then x+1 else x*2) [true;false;false;true] [3;4;5;6];;
- : int list = [4; 8; 10; 7]

# map2 pair [1;2;3;4] ['a';'b';'c'];;
- : (int * char) list = [(1, 'a'); (2, 'b'); (3, '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. OCAMLprovides 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>
```

2.2 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 lenght 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:

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

```
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];;
- : int list = [6; 4; 8]

# filter ((flip (>)) 4) [6;4;3;5;8;7;1];;
- : int list = [6; 5; 8; 7]

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

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

2.3 List Accumulation: Folding

2.3.1 `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/conque/glue is an incredibly effective strategy for process-

ing 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).

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 ns = foldr ( * ) 1 ns;;
val prod : int list -> int = <fun>

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

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

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

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

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

# let all_even ns = foldr (fun x ans -> ((x mod 2) = 0) && ans) true ns;;
val all_even : int list -> bool = <fun>

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

# let exists_positive ns = foldr (fun x ans -> (x > 0) || ans) false ns;;
val exists_positive : int list -> bool = <fun>

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

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

# let exists_even ns = foldr (fun x ans -> ((x mod 2) = 0) || ans) false ns;;
val exists_even : int list -> bool = <fun>

# exists_even [6;4;3;5;8;7;1];;
- : bool = true
```

Figure 2: `foldr` examples, part 1

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

# let append xs ys = foldr cons ys xs;;
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 append [] xss;; (* could also write foldr append (@) xss *)
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 (fun ys ans -> (x::ys) :: ans) [] yss;;
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 (fun x ans -> ans @ (mapcons x ans)) [[]] xs;;
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 (fun x ans -> (f x) :: ans) [] xs;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>

# let filter p xs = foldr (fun x ans -> if p x then x :: ans else ans) [] xs;;
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
```

Figure 3: `foldr` examples, part 2

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

Unfortunately, in some cases eta reduction interacts badly with OCAML's type reconstruction. For example, consider this alternative defintion 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. 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]
```

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 append [] xss;; (* could also write foldr append (@) 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']
```

2.3.2 `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 ((flip (>)) 0);;
val all_positive : int list -> bool = <fun>
# let all_even = for_all (fun x -> (x mod 2) == 0);;
val all_even : int list -> bool = <fun>
# all_positive [6;4;3;5;8;7;1];;
- : bool = true
# all_even [6;4;3;5;8;7;1];;
- : 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 3

# 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! In these cases, it's better to hand-craft versions of these functions 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'
```

2.3.3 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 followingtemplate:

```
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

Need more discussion of isSorted

The `isSorted` 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 o`isSorted [1;3;7;4;9]` is `false`. Can we define `isSorted` using `foldr` and friends? Yup! Here is one such defintion:

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

But there are other strategies. 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:[4]

```
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>
```

insert

2.3.4 `foldr'`

Need discussion here.

```
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

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

2.3.5 `foldr2` and Friends

Need discussion here.

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>
```

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

2.4 List Accumulation: `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
```

[4]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.

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.

Need more discussion here.

2.5 List Generation: `gen`

In addition to transforming lists, there are useful abstractions for producing and consuming 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;;
- : int list = [7; 8; 9; 10; 11; 12; 13; 14; 15; 16; 17; 18; 19]

# gen ((flip (/)) 2) ((=) 0) 100;;
- : int list = [100; 50; 25; 12; 6; 3; 1]

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

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;;
- : (int * int) list = [(5, 1); (4, 5); (3, 20); (2, 60); (1, 120)]

# 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;;
- : (int * int) list = [(0, 1); (1, 1); (1, 2); (2, 3); (3, 5); (5, 8); (8, 13)]

# map fst (fibsTo 100);;
- : int list = [0; 1; 1; 2; 3; 5; 8; 13; 21; 34; 55; 89]
```

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 iterate next isDone state =
  if isDone state then
    state
  else iterative 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 (x,a,b) -> (x-1,b,a+b)) (fun (x,_,_) -> x = 0) (n,0,1) with
      (_,ans,_) -> ans
val fibi : int -> int = <fun>

# fibi 10;;
- : int = 55
```

2.6 List Generation: `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>
```

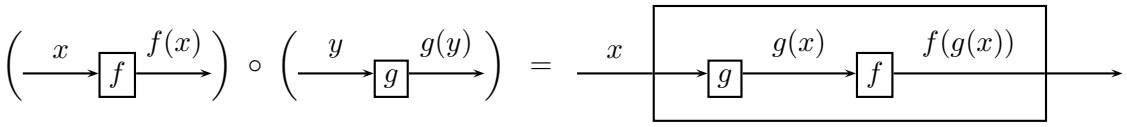
Need more discussion

3 Function Composition

Just as there are standard ways of combining two integers to yield another integer (e.g., `+` and `*`) and standard ways of combining two booleans to yield a boolean (e.g., `&&`, `||`), there are standard ways of combining two functions to yield a another function. The most important of these is **function composition**. In mathematics, if f and g are two functions, then the composition of f and g, written $f \circ g$, is defined as follows:

$$(f \circ g)(x) = f(g(x))$$

If we depict functions as boxes that take their inputs from their left and produce their outputs to the right, composition would be depicted as follows:



Note that the left-to-right nature of the graphical depiction of the function boxes requires inverting the order of the function boxes when they are composed. In contrast, the right-to-left nature of the textual notation requires no inversion.

Composition is straightforward to define in OCAML:

```
let o f g = fun x -> f (g x)
val o : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

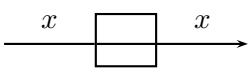
Here we have defined `o` as a curried prefix composition operator. Not that we could have also defined it without any explicit `fun` via `let o f g x = f (g x)`. Here are some examples involving composition:

```
# let inc = (+) 1;;
val inc : int -> int = <fun>

# let dbl = ( * ) 2;;
val dbl : int -> int = <fun>

# (o inc dbl) 10;;
- : int = 21

# (o dbl inc) 10;;
- : int = 22
```

Just as addition, multiplication, conjunction, and disjunction all have identity values (respectively, `0`, `1`, `true`, and `false`), so too does composition have an identity value — the identity function:

```
let id x = x
```

Graphically, the identity function is a function box that passes its argument unaltered:



You should convince yourself that `(o f id)` and `(o id f)` are functions that are behaviorally indistinguishable from `f`. This is easy to see from the graphical representation:

It is common to compose functions with themselves. For example:

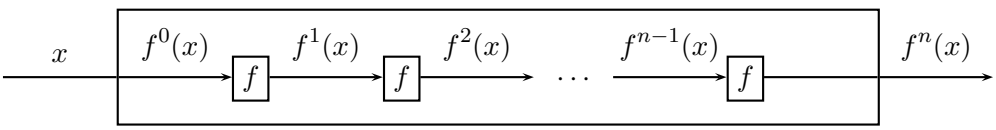
```
# let twice f = o f f;;
val twice : ('a -> 'a) -> 'a -> 'a = <fun>
(* (twice f) behaves like fun x -> f (f x) *)

# let thrice f = o f (twice f);;
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
(* (thrice f) behaves like fun x -> f (f (f x)) *)
```

Numerous examples involving `twice` and `thrice` are shown in Fig. 4.

```
# (twice inc) 10;; (* equivalent to inc (inc 10) *)
- : int = 12

# (twice dbl) 10;; (* equivalent to dbl (dbl 10) *)
- : int = 40

# (thrice inc) 10;; (* equivalent to inc (inc (inc 10)) *)
- : int = 13

# (thrice dbl) 10;; (* equivalent to dbl (dbl (dbl 10)) *)
- : int = 80

# (twice (twice inc)) 0;; (* equivalent to (twice inc) ((twice inc) 0) *)
- : int = 4

# (twice (thrice inc)) 0;; (* equivalent to (thrice inc) ((thrice inc) 0) *)
- : int = 6

# (thrice (twice inc)) 0;;
(* equivalent to (twice inc) ((twice inc) ((twice inc) 0)) *)
- : int = 6

# (thrice (thrice inc)) 0;;
(* equivalent to (thrice inc) ((thrice inc) ((thrice inc) 0)) *)
- : int = 9

# ((twice twice) inc) 0;; (* equivalent to (twice (twice inc)) 0 *)
- : int = 4

# ((twice thrice) inc) 0;; (* equivalent to (thrice (thrice inc)) 0 *)
- : int = 9

# ((thrice twice) inc) 0;; (* equivalent to (twice (twice (twice inc))) 0 *)
- : int = 8

# ((thrice thrice) inc) 0;; (* equivalent to (thrice (thrice (thrice inc))) 0 *)
- : int = 27
```

Figure 4: Examples involving the `twice` and `thrice` functions.

More generally, the n-fold composition of a function f, written f^n, is the result of composing n copies of f. (f^0, the zero-fold composition of f , is just the identity function.) Here is a graphical depiction of f^n :



In OCAML,, n-fold composition can be expressed via the following `n_fold` function:

```
let rec n_fold n f =
  if n = 0 then
    id
  else
    o f (n_fold (n-1) f)
```

For example:

```
# n_fold 5 inc 0;;
- : int = 5

# n_fold 3 dbl 1;;
- : int = 8

# n_fold 3 (fun x -> x * x) 2;;
- : int = 256

# n_fold 0 (fun x -> x * x) 17;;
- : int = 17
```

Note that the `twice` and `thrice` functions from above can be defined in terms of `n_fold`:

```
# let twice f = n_fold 2 f;;
val twice : ('a -> 'a) -> 'a -> 'a = <fun>

# let thrice f = n_fold 3 f;;
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
```

4 Church Numerals

The celebrated logician Alonzo Church (who invented the lambda calculus, a formal mathematical system upon which functional programming is based) observed that n-fold composition functions can be viewed as numerals in the sense that it is possible to perform arithmetic on them (e.g. addition, multiplication, exponentiation, etc.). Such numerals are called Church numerals in his honor. We shall use `int2ch` and `ch2int` to convert between OCAML integers and church numerals:

```
# let int2ch = n_fold (* synonym for n_fold *)
val int2ch : int -> ('a -> 'a) -> 'a -> 'a = <fun>

# let ch2int c = c ((+) 1) 0
val ch2int : ((int -> int) -> int -> 'a) -> 'a = <fun>
```

`ch2int` finds the integer corresponding to an n-fold composition function by incrementing n times starting at 0. For example:

```
# ((int2ch 17) inc) 10;;
- : int = 27

# ch2int (int2ch 17);;
- : int = 17

# ch2int twice;;
- : int = 2

# ch2int thrice;;
- : int = 3

# let nonce = fun f x -> x;;
val nonce : 'a -> 'b -> 'b = <fun>
```

```
# ch2int nonce;;
- : int = 0

# let once = fun f x -> f x;;
val once : ('a -> 'b) -> 'a -> 'b = <fun>

# ch2int once;;
- : int = 1
```

As an example of arithmetic on Church numerals, consider the successor function `succ` that adds one to a Church numeral. Since the Church numeral corresponding to n is the n-fold composition function, applying `succ` to such a church numeral should yield an $(n+1)$-fold composition function. Here's one definition:

```
# let succ c = fun f -> o f (c f)
val succ : (('a -> 'b) -> 'c -> 'a) -> ('a -> 'b) -> 'c -> 'b = <fun>

# ch2int (succ twice);;
- : int = 3

# ch2int (succ thrice);;
- : int = 4

# ch2int (succ nonce);;
- : int = 1

# ch2int (succ once);;
- : int = 2

# (succ thrice) dbl 1;;
- : int = 16
```

Note that an alternative definition of `succ` would be:

```
#  let succ c = fun f -> o (c f) f;;
val succ : (('a -> 'b) -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c = <fun>
```

It is also possible to define a function `plus` that adds two Church numerals. One definition of `plus` is based on the following observation: if `c1` and `c2` are Church numerals for n_1 and n_2, then `(c1 f)` performs `f` n_1 times, `c2` performs `f` n_2 times, and `(o (c1 f) (c2 f))` performs `f` $n_1 + n_2$ times.

```
# let plus c1 c2 = fun f -> o (c1 f) (c2 f);;
val plus : ('a -> 'b -> 'c) -> ('a -> 'd -> 'b) -> 'a -> 'd -> 'c = <fun>

# (ch2int (plus once nonce));;
- : int = 1

# (ch2int (plus once nonce));;
- : int = 5
```

An alternative definition of `plus` is based on the observation that addition is repeated incrementing:

```
# let plus' c1 c2 = ((c1 succ) c2);;
val plus' :
  ((((('a -> 'b) -> 'c -> 'a) -> ('a -> 'b) -> 'c -> 'b) -> 'd -> 'e) ->
  'd -> 'e = <fun>

# (ch2int (plus' (int2ch 2) (int2ch 3)));;
- : int = 5
```

```
# (ch2int (plus' (int2ch 1) (int2ch 0)));;
- : int = 1
```

In a similar manner, it is possible to define functions `times` and `expt` that perform multiplication and exponentiation on Church numerals. [5] Example calls of these functions are shown below, but their definition is left as an exercise. (*Hints:* (1) carefully study the `twice`/`thrice` examples from above; (2) think in terms of the function box notation.)

```
# (ch2int (times (int2ch 2) (int2ch 3)));;
- : int = 6

# (ch2int (times (int2ch 3) (int2ch 2)));;
- : int = 6

# (ch2int (expt (int2ch 2) (int2ch 3)));;
- : int = 8

# (ch2int (expt (int2ch 3) (int2ch 2)));;
- : int = 9
```

It is also possible to define a `pred` function that decrements a Church numeral (and acts as the identity on the Church numeral for 0):

```
# (ch2int (pred (int2ch 3));;
- : int = 2

# (ch2int (pred (int2ch 0));;
- : int = 0
```

[5] These functions can be defined *without* using `int2ch`, `ch2int`, `n_fold`, or any recursively defined OCAML functions.