CS251 Programming Languages **Handout # 13**
Randy Shull
Wellesley College

First-Class Functions

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 Naming Functions

By the naming property of first-class functions, 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);;
⇒ (fun (a,b) -> (a+b)/2) (8,10)
⇒ (8+10)/2
⇒ 9
- : 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`.

1.2 Passing Functions as Arguments

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

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

```
# app_3_5' (fun (x,y) -> x + y);;
⇒ (fun f -> f (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 f -> f (3,5)) (fun (x,y) -> x * y)
⇒ (fun (x,y) -> x * y) (3,5)
⇒ 3 * 5
⇒ 15
```

```
# app_3_5' avg
⇒ (fun f -> f (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 f -> f (3,5)) (fun (a,b) -> a)
⇒ (fun (a,b) -> a) (3,5)
⇒ 3

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

1.3 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 b to the power p.

```
# let to_the = fun p -> (fun b -> expt(b,p))
(* Top-level environment now contains binding
   to_the ↦ 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;;
⇒ (fun f -> f 3 5) (fun p -> fun b -> expt(b,p))
⇒ (fun p -> fun b -> expt(b,p)) 3 5
⇒ expt(5,3)
- : 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` that is equivalent to `fun x y -> x+y`:

```
# (+) 1 2;;
⇒ (fun x y -> x+y) 1 2
⇒ 1+2
- : int = 3
```

We can use these with `app_3_5`:

```
# app_3_5 (+);;
⇒ (fun f -> f 3 5) (fun x y -> x+y)
⇒ (fun x y -> x+y) 3 5
⇒ 3+5
- : int = 8
```

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

```
# app_3_5 (-);;
⇒ (fun f -> f 3 5) (fun x y -> x-y)
⇒ (fun x y -> x-y) 3 5
⇒ 3-5
- : int = -2
```

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

A key benefit to using curried functions (as opposed to tupled functions) is that partially applied curried functions are useful in many situations. For example:

```
# let app5 = fun f -> f 5;;
val app5 : (int -> 'a) -> 'a = <fun>
```

```
# app5 (to_the 2)
⇒ (fun f -> f 5) ((fun p -> fun b -> expt(b,p)) 2)
⇒ (fun f -> f 5) (fun b -> expt(b,2))
⇒ (fun b -> expt(b,2)) 5
⇒ expt(5,2)
- : int = 25
```

```
# app5 ((+) 1)
⇒ (fun f -> f 5) ((fun x y -> x+y) 1)
⇒ (fun f -> f 5) (fun y -> 1+y)
⇒ (fun y -> 1+y) 5
⇒ 1+5
- : int = 6
```

```
# app5 ((-) 1)
⇒ (fun f -> f 5) ((fun x y -> x-y) 1)
⇒ (fun f -> f 5) (fun y -> 1-y)
⇒ (fun y -> 1-y) 5
⇒ 1-5
- : int = -4
```

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 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;;
⇒ ((fun f a b -> f b a) (fun x y -> x-y)) 3 2
⇒ (fun a b -> (fun x y -> x-y) b a) 3 2
⇒ (fun x y -> x-y) 2 3
⇒ 2-3
- : int = -1
```

```
# (flip to_the) 5 2;;
⇒ ((fun f a b -> f b a) (fun p -> fun b -> expt(b,p))) 5 2
⇒ (fun a b -> (fun p b -> expt(b,p)) b a) 5 2
⇒ (fun p b -> expt(b,p)) 2 5
⇒ expt(5,2)
- : int = 25
```

```
# app_3_5 (flip (<));;
⇒ (fun f -> f 3 5) ((fun f a b -> f b a) (fun x y -> x<y))
⇒ (fun f -> f 3 5) (fun a b -> (fun x y -> x<y) b a)
⇒ (fun a b -> (fun x y -> x<y) b a) 3 5
⇒ (fun x y -> x<y) 5 3
⇒ 5<3
- : bool = false
```

```
# app5 ((flip (-)) 1)
⇒ (fun f -> f 5) ((fun f a b -> f b a) (fun x y -> x-y) 1)
⇒ (fun f -> f 5) (fun b -> (fun x y -> x-y) b 1)
⇒ (fun b -> (fun x y -> x-y) b 1) 5
⇒ (fun x y -> x-y) 5 1
⇒ 5-1
- : int = 4
```

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
val church_pair : 'a -> 'b -> ('a -> 'b -> 'c) -> 'c = <fun>
```

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

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

functional "pair":

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

```
# church_fst (church_pair 17 32);;
⇒ (fun cp -> cp (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.

1.4 Storing Functions in Data Structures

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

```
# let fun_tuple = (to_the, (<), app_3_5, church_pair);;
val fun_tuple :
  (int -> int -> int) * (int -> int -> bool) * ((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 2 1 g);;
⇒ match (to_the, (<), app_3_5, church_pair)
    with (f,g,h,k) -> (h f, k 2 1 g);;
⇒ (app_3_5 to_the, church_pair 2 1 (<))
⇒ ((fun f -> f 3 5) (fun p b -> expt(b,p)),
    (fun x y f -> f x y) 2 1 (fun x y -> x<y))
⇒ ((fun p b -> expt(b,p)) 3 5, (fun x y -> x<y) 2 1)
⇒ (expt(5,3), 2<1)
- : int * bool = (125, false)
```

1.5 Creating Functions

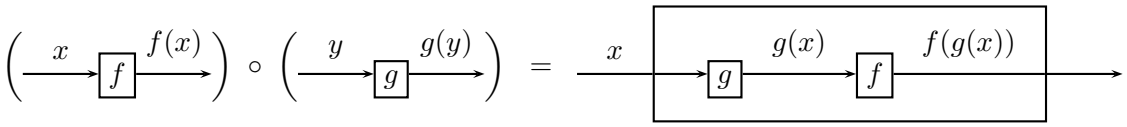
Finally, functions can be created in any context. In many programming languages, such as C, functions can only be defined at "top-level"; it is not possible to declare one function inside of another function. But as seen above in the `to_the` and `church_pair` examples, the ability to specialize a function 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 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:

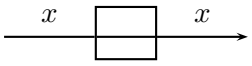
```
# let inc = (+) 1;;
⇒ let inc = (fun x y -> x+y) 1
⇒ let inc = (fun y -> 1+y)
val inc : int -> int = <fun>

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

# (o inc dbl) 10;;
⇒ (fun f g x -> f (g x)) inc dbl 10
⇒ inc (dbl 10)
⇒ (fun y -> 1+y) ((fun y -> 2*y) 10)
⇒ (fun y -> 1+y) (2*10)
⇒ (fun y -> 1+y) 20
⇒ 1+20
- : int = 21
```

```
# (o dbl inc) 10;;
⇒ (fun f g x -> f (g x)) dbl inc 10
⇒ dbl (inc 10)
⇒ (fun y -> 2*y) ((fun y -> 1+y) 10)
⇒ (fun y -> 2*y) (1+10)
⇒ (fun y -> 2*y) 11
⇒ 2*11
- : int = 22
```

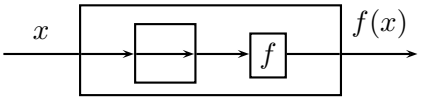
Just as addition (`+`), multiplication (`*`), conjunction (`&&`), and disjunction (`||`) all have identity values (respectively, `0`, `1`, `true`, and `false`), so too does composition have an identity value — the identity function:

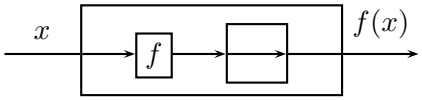
```
let id x = x
```

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:

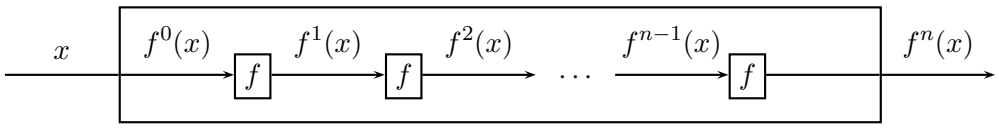
```
# let twice f = o f f;;
val twice : ('a -> 'a) -> 'a -> 'a = <fun>
(* (twice f) behaves like fun x -> f (f x) *)

# 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. 1.

```
# (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 1: 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>
```