# Lecture 2: OCaml Functions

CS 6371: Advanced Programming Languages
January 20, 2022

```
#let solve w x y z =
    let prod a b = a*b in
    (prod w x)+(prod y z);;
solve : int -> int -> int -> int -> int =<fun>
#prod;;
Toplevel input:
>prod;;
>^^^^
The value identifier prod is unbound.
```

Top-level and nested “let” operations are different: A top-level “let” defines a global variable (usually a function). It must be followed by “;;”. An inner “let” declares a local variable that can be used only within a subexpression. It must be followed by “in”. When variable names conflict, the innermost declaration “shadows” the others.

```
#type foo = int;;
#type foo = bool;;
#type foo = string;;
#type foo = int list;;
#type foo = int * string;;
#type btree = BNil
    | BNode of (int*btree*btree);;
#type ntree = NNil
    | NNode of ((int*ntree) list);;
```

User-defined types can be primitive types (int, bool, string, etc.), or they can be lists, tuples, or variants that include any of the above.

```
#let identity x = x;;
identity : 'a -> 'a = <fun>
#identity 3;;
- : int = 3
#identity "foo";;
- : string = "foo"
```

When possible, OCaml gives functions a polymorphic type. Polymorphic functions can be applied to arguments of any type.

```
#let apply f x = (f x);;
apply : ('a -> 'b) -> 'a -> 'b =<fun>
#let add (x,y) = x+y;;
add : int * int -> int =<fun>
#apply add (1,2);;
- : int = 3
#apply add "foo";;
Toplevel input:
>apply add "foo";;
>          ^^^^^
```

This expression has type string, but is used with type int * int.

```
#let rec map f l =
    (match l with
     | [] -> []
     | x::t -> (f x)::(map f t));;
map :('a -> 'b) -> 'a list -> 'b list =<fun>
#let addone n = n+1;;
addone : int -> int =<fun>
#map addone [23;42;64];;
- : int list = [24; 43; 65]
```

Lists can also have polymorphic type.

```
#map (fun n -> n+1) [23;42;64];;
- : int list = [24; 43; 65]
#(fun n -> n+1) 2;;
- : int = 3
```

Lists can also have polymorphic type. Use “fun” to create anonymous (i.e., unnamed) functions. “fun ... -> ...” is the same as if you typed “let foo ... = ...;;” and then used “foo”.

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```
Using anonymous functions, you can build and return functions as values at runtime.

<table>
<thead>
<tr>
<th>Anonymous functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>compose</code></td>
<td>Builds and returns a function as a value at runtime.</td>
</tr>
<tr>
<td><code>cool</code></td>
<td>An anonymous function that refers to variables declared in outer scopes.</td>
</tr>
<tr>
<td><code>addx</code></td>
<td>An anonymous function with a typing annotation to help debug.</td>
</tr>
<tr>
<td><code>add</code></td>
<td>Actually “let foo x y = …” is just an abbreviation for “let foo = (fun x -&gt; (fun y -&gt; ...))”.</td>
</tr>
<tr>
<td><code>map2</code></td>
<td>Binary operators can be used in prefix rather than infix syntax by enclosing them in parentheses.</td>
</tr>
</tbody>
</table>

An anonymous function may refer to variables declared in outer scopes.

Actually “let foo x y = ...” is just an abbreviation for “let foo = (fun x -> (fun y -> ...))”. If you give such a function fewer arguments than it expects, it yields a function from the remaining arguments to the original value. Functions written this way are called “curried functions”. Applying fewer arguments is called “partial evaluation”.

Typing annotations are almost never necessary, but they can help you debug.
You can also use a typing annotation to restrict the type of a function that would otherwise be polymorphic.

Typing annotations may contain type variables.

A ;-separated sequence of expressions is evaluated in order. The last expression is returned as the result of the sequence expression. Use sequence expressions with print statements to debug.

OCaml’s equality operator (=) tests structural equality. This means that you can use it with ints, bools, strings, tuples, lists, and variants.