#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.

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.

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

However, there must be some consistent way to instantiate each type variable. Here we see an example where no such instantiation exists and the compiler therefore rejects the code.

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”.

<table>
<thead>
<tr>
<th>Lecture 2: OCaml Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS 6371: Advanced Programming Languages</td>
</tr>
<tr>
<td>January 16, 2020</td>
</tr>
</tbody>
</table>

```ocaml
define solve : w x y z =  
  (let prod a b = a*b in  
   (prod w x)+(prod y z);;  
  )

#prod;;  
Toplevel input:  
>prod;;  
>^^^^  
The value identifier prod is unbound.
```

```ocaml

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

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

#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 "foo";;  
Toplevel input:  
>apply "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]

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

```
```
```ml
#let compose f g = (fun x -> f (g x));;
compose : ('a->'b)->('c->'a)->'c->'b = <fun>

#let cool = (compose (fun n -> n+1)
  (fun n -> n*2));;
cool : int -> int = <fun>

#let addx x = (fun y -> x+y));;
addx : int -> int -> int = <fun>

#let add = (fun x -> (fun y -> x+y));;
add : int -> int -> int = <fun>

#let rec map2 f l1 l2 =
  (match (l1,l2) with
    (\[],x) -> x
  | (x,\[]) -> x
  | (h1::t1, h2::t2) ->
    (f h1 h2)::(map2 f t1 t2));;
map2 : ('a -> 'a -> 'a) -> 'a list -> 'a list -> 'a list = <fun>

#let rec addpairs l =
  (match l with
    x::y::t -> (add x)::(addpairs t)
    | [] -> []
    | [x] -> [x]);;
Toplevel input:
>   x::y::t -> (add x)::(addpairs t)
>                       ^^^^^^^^^^^^
This expression has type int list, but is used with type (int -> int) list.
```

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

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”.

Binary operators can be used in prefix rather than infix syntax by enclosing the operator in parentheses. This allows you to pass a binary operator as a function argument.

Typing annotations are almost never necessary, but they can help you debug.
<table>
<thead>
<tr>
<th>Code</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#let intident (x:int) = x;;</code> intident : int -&gt; int = &lt;fun&gt;</td>
<td>You can also use a typing annotation to restrict the type of a function that would otherwise be polymorphic.</td>
</tr>
<tr>
<td><code>#let apply (f:'inp-&gt;'out) (x:'inp) = (f x);;</code> apply : ('a -&gt; 'b) -&gt; 'a -&gt; 'b = &lt;fun&gt;</td>
<td>Typing annotations may contain type variables.</td>
</tr>
<tr>
<td><code>#let main () = (print_string &quot;hello\n&quot;; print_int 3; print_newline (); 12);;</code> main : unit -&gt; int = &lt;fun&gt;</td>
<td>A <code>;</code>-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.</td>
</tr>
<tr>
<td><code>#&quot;foo&quot; = &quot;foo&quot;;;</code> - : bool = true</td>
<td>OCaml’s equality operator (=) tests structural equality. This means that you can use it with ints, bools, strings, tuples, lists, and variants.</td>
</tr>
<tr>
<td><code>#(3,&quot;foo&quot;) = (3,&quot;foo&quot;);;</code> - : bool = true</td>
<td></td>
</tr>
<tr>
<td><code>#[1;2;3] = [1;2;3];;</code> - : bool = true</td>
<td></td>
</tr>
<tr>
<td><code>#Dark (Dark Red) = Dark (Dark Red);;</code> - : bool = true</td>
<td></td>
</tr>
</tbody>
</table>