Lecture 2: OCaml Functions

CS 6371: Advanced Programming Languages
January 19, 2023

```ocaml
#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;;
Top-level input:
>prod;;
>^^^^
The value identifier prod is unbound.
```

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.

```ocaml
#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);;

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

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

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.

```ocaml
#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.

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

```ocaml
#map (fun n -> n+1) [23;42;64];;
- : int list = [24; 43; 65]
#(fun n -> n+1) 2;;
- : int = 3
```
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>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#let intident (x:int) = x;;</code>&lt;br&gt;<code>intident : int -&gt; int = &lt;fun&gt;</code></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>&lt;br&gt;<code>apply : ('a -&gt; 'b) -&gt; 'a -&gt; 'b = &lt;fun&gt;</code></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>&lt;br&gt;<code>main : unit -&gt; int = &lt;fun&gt;</code>&lt;br&gt;<code>#main ();;</code>&lt;br&gt;<code>hello 3</code>&lt;br&gt;<code>- : int = 12</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>#&quot;foo&quot; = &quot;foo&quot;;;</code>&lt;br&gt;<code>- : bool = true</code>&lt;br&gt;<code>#(3,&quot;foo&quot;) = (3,&quot;foo&quot;);;</code>&lt;br&gt;<code>- : bool = true</code>&lt;br&gt;<code>#[1;2;3] = [1;2;3];;</code>&lt;br&gt;<code>- : bool = true</code>&lt;br&gt;<code>#Dark (Dark Red) = Dark (Dark Red);;</code>&lt;br&gt;<code>- : bool = true</code></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>
</tbody>
</table>