

Lecture Notes: Introduction to OCaml

CS 6371: Advanced Programming Languages

August 20, 2007

| | |
|---|--|
| <pre>#1+1;; - : int = 2 #1+2*3;; - : int = 7</pre> | OCaml has a built-in type "int" that supports the usual binary operators. |
| <pre>#let add x y = x+y;; add : int -> int -> int = <fun></pre> | Use "let" to define a function. OCaml responds by telling you the "type" of the new function you've created. This one is a function from two integers to an integer. |
| <pre>#add 3 4;; - : int = 7</pre> | Instead of applying a function with syntax "f(x,y)", OCaml uses syntax "(f x y)". |
| <pre>#let hypotenuse x y = let xsquared = x*x in let ysquared = y*y in (xsquared + ysquared);; hypotenuse : int -> int -> int = <fun></pre> | "let ... in ..." can be used within a function definition to declare variables and assign them values. Note that a variable's definition never changes! It is assigned exactly once. |
| <pre>#if 3<4 then (add 1 2) else (add 5 6);; - : int = 3</pre> | In OCaml, "if...then...else..." is an expression not a command. It's like "... ? ... : ..." in C or Java. |
| <pre>#let test x = if x<4 then "yes" else 0;; Toplevel input: >let test x = if x<4 then "yes" else 0;; > This expression has type int, but is used with type string.</pre> | The two branches of the "if" must return values of the same type. The example produces an error because one branch returns a string while the other returns an int. |
| <pre>#true;; - : bool = true #false;; - : bool = false #true && false;; - : bool = false #false false;; - : bool = false</pre> | In addition to integers and strings, OCaml also has booleans. Conjunction is "&&" and disjunction is " " just like in C or Java. Unlike C, booleans and integers are not interchangeable! |
| <pre>#"foo" ^ "bar";; - : string = "foobar"</pre> | The "^" operator performs string concatenation. |
| <pre>#let rec factorial n = if n<=1 then 1 else n*(factorial (n-1));; factorial : int -> int = <fun></pre> | A "recursive function" calls itself. To define a recursive function, put "rec" after the "let". |
| <pre>#type color = Red Blue Dark of color Light of color;; Type color defined. #Red;; - : color = Red #Dark Blue;; - : color = Dark Blue #Light (Dark Blue);; - : color = Light (Dark Blue)</pre> | In OCaml you can define your own types with the "type" directive. In this type, "Red", "Blue", "Dark", and "Light" are the "type constructors" for type "color". |
| <pre>#Light Dark Blue;; Toplevel input: >Light Dark Blue;; > This expression has type color -> color, but is used with type color.</pre> | Notice that I used parentheses in the last example. If I hadn't, an error would have resulted. This is because type constructors associate left by default. |

| | |
|---|--|
| <pre>#let isred c = (match c with Red -> true x -> false);; isred : color -> bool = <fun> #let isdark c = (match c with Dark x -> true x -> false);; #let rec isred c = (match c with Red -> true Dark x -> isred x Light x -> isred x x -> false);; isred : color -> bool = <fun></pre> | <p>The “match ... with ...” operator allows you to test whether a value matches a type constructor. The left side of each -> is called a “pattern”. Patterns can contain variables. If the pattern matches, the variables become bound to the respective parts of the value being tested and may be used with the right-hand side of the ->.</p> |
| <pre>#let rec isred c = (match c with Red -> true Dark x -> isred x Light x -> isred x _ -> false);; isred : color -> bool = <fun> #isred (Dark Red);; - : bool = true</pre> | <p>Anywhere you would normally put a variable in a pattern you can instead put an underscore. Underscore matches to anything (just like a variable) except that it doesn’t bind any variable to the matching sub-expression.</p> |
| <pre>#let mylist = [4; 8; 15; 16; 23];; mylist : int list = [4; 8; 15; 16; 23] #0::mylist;; - : int list = [0; 4; 8; 15; 16; 23] #0::1::mylist;; - : int list = [0; 1; 4; 8; 15; 16; 23]</pre> | <p>OCaml has a list type. Lists are enclosed in brackets and elements are separated by semicolons. The :: operator (called “cons”) inserts an element onto the head of a list.</p> |
| <pre>>["foo"; 3];; >^^^^^^^^^^ This expression has type int list, but is used with type string list.</pre> | <p>All elements of a list must have the same type.</p> |
| <pre>#let rec length list = (match list with [] -> 0 x::tail -> (length tail)+1);; length : 'a list -> int = <fun> #let rec addpairs list = (match list with [] -> [] x::[] -> [x] x::y::t -> (x+y)::(addpairs t));; addpairs : int list -> int list = <fun></pre> | <p>You can use “match” to match lists. The pattern “[]” matches the empty list. Pattern “x::tail” matches a list with at least one element. Pattern “x::y::tail” matches a list with at least two elements, etc.</p> |
| <pre>#("foo",3);; - : string * int = "foo", 3</pre> | <p>A “tuple” is a fixed-length collection of values. The members of the collection need not have the same type. This is an example of a string-int pair.</p> |
| <pre>#let math x y = (x+y, x-y, x*y);; math : int -> int -> int * int * int = <fun></pre> | <p>Tuples are useful when you want to return more than one value from a function.</p> |
| <pre>#let (sum,diff,prod) = (math 2 3);; sum : int = 5 diff : int = -1 prod : int = 6 #let add (x,y) = x+y;; add : int * int -> int = <fun> #match (math 2 3) with (sum,_,_) -> sum;; - : int = 5</pre> | <p>You can “project” (i.e., pull apart) a tuple using “let” or “match”.</p> |
| <pre>#();; - : unit = () #let main () = "hello world";; main : unit -> string = <fun> #main ();; - : string = "hello world"</pre> | <p>The tuple with zero elements is called “unit”. It is useful when you don’t want to pass anything to a function.</p> |