Lecture 1: Introduction to OCaml

CS 4301/6371: Advanced Programming Languages
January 18, 2024

OCaml has a built-in type “int” that supports the usual binary operators.

```
#1+1;;
- : int = 2
#1+2*3;;
- : int = 7
```

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.

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

Instead of applying a function with syntax “f(x,y)”, OCaml uses syntax “(f x y)”.

```
#let hypotenuse x y =
    let xsquared = x*x in
    let ysquared = y*y in
    (xsquared + ysquared);;
hypotenuse : int -> int -> int = <fun>
```

“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.

```
#if 3<4 then (add 1 2) else (add 5 6);;
- : int = 3
```

In OCaml, “if…then…else…” is an expression not a command. It’s like “… ? … : …” in C or Java.

```
#let test x = if x<4 then "yes" else 0;;
```

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

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!

```
#"foo" ^ "bar";;
- : string = "foobar"
```

The “^” operator performs string concatenation.

A “recursive function” calls itself. To define a recursive function, put “rec” after the “let”.

```
#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)
```

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”.

```
#Light Dark Blue;;
```

Toplevel input:
>Light Dark Blue;;
> ^^^^  # Light (Dark Blue)
```

This expression has type color -> color, but is used with type color.

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.
| #let isred c = (match c with Red -> true | x -> false);; | 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 ->. |
| isred : color -> bool = <fun> | |
| #let isdark c = (match c with Dark x -> true | x -> false);; |
| isdark : color -> bool = <fun> | 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. |
| #let rec isred c = (match c with Red -> true | Dark x -> isred x | Light x -> isred x | x -> false);; |
| isred : color -> bool = <fun> | 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. |
| #let rec isred c = (match c with Red -> true | Dark x -> isred x | Light x -> isred x | _ -> false);; #isred (Dark Red);; |
| - : bool = true | All elements of a list must have the same type. |
| #let mylist = [4; 8; 15; 16; 23];; mylist : int list = [4; 8; 15; 16; 23] | You can use “match” to match lists. The pattern “[]” matches the empty list. Pattern “a::b” matches a list with at least one element. Pattern “a::b::c” matches a list with at least two elements, etc. |
| #0::mylist;; - : int list = [0; 4; 8; 15; 16; 23] #0::1::mylist;; - : int list = [0; 1; 4; 8; 15; 16; 23] | |
| >("foo",3);; "foo", 3 This expression has type int list, but is used with type string list. |
| #let rec length s = (match s with [] -> 0 | x::t -> (length t)+1);; length : 'a list -> int = <fun> | 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. |
| #let rec addpairs s = (match s with [] -> [] | x::[] -> [x] | x::y::t -> (x+y)::(addpairs t));; addpairs : int list -> int list = <fun> | Tuples are useful when you want to return more than one value from a function. |
| #("foo",3);; - : string * int = "foo", 3 | You can “project” (i.e., pull apart) a tuple using “let” or “match”. |
| #let math x y = (x+y, x-y, x*y);} math : int -> int -> int * int * int = <fun> | The tuple with zero elements is called “unit”. It is useful when you don’t want to pass anything to a function. |
| #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 | |
| #();; - : unit = () #let main () = "hello world";; main : unit -> string = <fun> | |
| #main ();; - : string = "hello world" | |