Lecture 20: OCaml, continued

10:00 AM, Oct 22, 2018

Contents

1 Precedence in OCaml

2 Review of matching in OCaml
   2.1 Curry
   2.2 Matching combinations of tuples and lists

3 Review of variant types

1 Precedence in OCaml

Quiz: Write the following expressions with parentheses added to indicate how they will be evaluated by OCaml. You can assume the identifiers have been bound to appropriate values.

Hints:

- Usual arithmetic precedence among infix operations +, -, *, /, **, and the float equivalents.
- Procedure application takes precedence over (binds tighter than) infix operators.
- In case of tie, leftmost first.

1. \( a + f \ x \rightarrow a + (f \ x) \)
2. \( f \ x \* a \rightarrow (f \ x) \* a \)
3. \( a::f \ x \rightarrow a::(f \ x) \)
4. \( f \ x::a \rightarrow (f \ x)::a \)

2 Review of matching in OCaml

Quiz: Suppose \( x \) and \( y \) are bound to a Boolean value. Without using OCaml’s built-in OR operation, which is infix | |, write an expression using match whose value is the logical OR of the two values (i.e. is true if one or both of \( x \) and \( y \) is true). Be as brief as you can.

```
match x, y with
  | true, _ -> true
  | _, true -> true
  | _ -> false
```
**Quiz:** Under the same rule as before, write a procedure `my_or` that takes a pair of Booleans and returns their OR. Also write the type (signature) for this procedure.

\[
(* \text{bool} \times \text{bool} -> \text{bool} *)
\]

```ocaml
let my_or = function (x, y) ->
  match (x, y) with
  | (true, _) -> true
  | (_, true) -> true
  | _ -> false
```

In the above solution we have indicated the procedure type in the comments, which are indicated as follows:

\[
(* \text{This is a comment in OCaml} *)
\]

Note that instead we can incorporate the type signature into the definition as a *type annotation*, which will help with debugging our code:

```ocaml
let my_or : bool * bool -> bool =
  function (x, y) ->
    match (x, y) with
    | (true, _) -> true
    | (_, true) -> true
    | _ -> false
```

Now you don’t need the comma, and OCaml will tell you if you are wrong.

**We will require as part of design recipe that OCaml code include a type signature, either in a comment or in the definition itself.**

### 2.1 Curry

The `my_or` procedure takes two arguments. Now let’s do a curried version. The curried version of a two argument procedure is a procedure that takes one argument and returns another one argument procedure that returns the or.

```ocaml
let curried_or =
  function x -> fun y ->
    match x, y with
    | (true, _) -> true
    | (_, true) -> true
    | _ -> false;;
```

Here’s how we can use it.
# (curried_or true) false;;
- : bool = true
# curried_or true false;;
- : bool = true

Note that you don’t need the parens because of the “leftmost wins in case of tie” precedence rule.

**Quiz:** What is the type of `curried_or`?

**Answer:** `bool -> (bool -> bool)`, where `(bool -> bool)` indicates the procedure that takes in a `bool` and returns a `bool`. So this type signature tells us that we are writing a procedure that takes in a `bool` and produces a procedure which takes in a `bool` and produces a `bool`. Note: OCaml will write this as `bool -> bool -> bool` (without the parentheses).

### 2.2 Matching combinations of tuples and lists

In the above examples, we have been using patterns that match pairs, but you can use patterns to match triples, quadruples, ..., any length tuples. You can also match lists of any specific length. You can combine patterns, e.g. tuples of lists or lists of tuples.

**Quiz:** Write a pattern that matches a triple of two-element lists.

**Answer:** `[a; b] , [c ; d] , [e ; f]`. There are multiple right answers to this question – any identifiers will do. Some other valid answers are:

- `[a::b] , [c::d] , [e::f]`
- Omitting any or all of these variables by using a wild card: 
  `[_; _] , [ _; _], [ _; _]`

Note: You can use any identifiers you like but beware of shadowing important identifiers like the formal arguments.

Can we match all triples in which the three elements are the same? You might think that a pattern `a,a,a` would work but you are not allowed to use the same variable more than once in a given pattern.

### 3 Review of variant types

**Quiz** Find and correct three syntax errors and one likely programming bug.

type suit_length = Short | Regular | Long | X_Long;;
type color = Red | Blue | Yellow;;
type mensware =
  Jacket of int * suit_length | Pants of int * int | Necktie color;;
type inventory = mensware list;;
let Inv:inventory =
  [Jacket (36, Short) ; Jacket (42 Long);
  Necktie Blue; Pants (29, 29); Necktie Red];;
let mine = function
  | Jacket (36, short) -> true
  | Pants (29, 29) -> true
  | Necktie _ -> true;;

Corrections:

- Necktie of color instead of “Necktie color”.
- short → Short (all types must be Capitalized)
- Jacket (42, Long) instead of Jacket (42 Long) (must include a comma)
- let inv:inventory = ... (Only constructors are allowed to start with capital letters).
- Pattern-matching in procedure mine should maybe have an else clause _ -> false.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: http://cs.brown.edu/courses/csci0170/feedback.