Multiple Choice: What does the following OCaml function do?

```
let foo x y = (match x with y -> "yes");;
```

(A) returns “yes” only when x=y
(B) compiles with an “inexhaustive match” warning
(C) always returns “yes”
(D) both A and B
Pattern Matches vs. Equality Tests

• (match exp with pat -> ... | pat -> ...)
  – Evaluates one expression and tests to see if the resulting value matches any of the patterns

• (if exp=exp then ... else ...)
  – Evaluates two expressions and tests to see if the resulting values are identical
  – Use single-equals “=” for equality tests in OCaml. Never use “==” in this class (tests “physical equality”, which means something different!).
# Patterns vs. Expressions

<table>
<thead>
<tr>
<th>Patterns</th>
<th>Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Never evaluated</strong></td>
<td><strong>Evaluated</strong></td>
</tr>
<tr>
<td>x</td>
<td>a pattern that matches everything, introducing a new variable x</td>
</tr>
<tr>
<td></td>
<td>an expression that evaluates to the value of existing variable x</td>
</tr>
<tr>
<td></td>
<td>not a valid expression</td>
</tr>
<tr>
<td>x::t</td>
<td>a pattern that matches any list with at least one element, introducing new variables x and t</td>
</tr>
<tr>
<td></td>
<td>an expression that creates a new list by cons’ing a copy of element x onto a copy of list t</td>
</tr>
<tr>
<td>3</td>
<td>a pattern that matches the integer value 3</td>
</tr>
<tr>
<td></td>
<td>an expression that returns the integer value 3</td>
</tr>
<tr>
<td>3+4</td>
<td>not a valid pattern</td>
</tr>
<tr>
<td></td>
<td>evaluates to 7</td>
</tr>
</tbody>
</table>
Currying

• **Def:** A function is *curried* if none of its arguments has a tuple type.
  – Curried functions have types of the form $\tau_1 \rightarrow \tau_2 \rightarrow \ldots \rightarrow \tau_n$
  – The arrow type operator is right-associative, so whenever we write the above, it means $\tau_1 \rightarrow (\tau_2 \rightarrow (\ldots \rightarrow \tau_n))$
  – Function application is left-associative, so $(\text{func} \ a1 \ a2 \ \ldots \ \text{an})$ is short for $(((\text{func} \ a1) \ a2) \ \ldots \ \text{an})$

• **Def:** To *curry* a function means to convert any tuple arguments into arrow arguments
  – Exercise: Curry the function “let add (x,y) = x+y;;”
  – Solution: let add x y = x+y;;
  – Another solution: let add = fun x -> fun y -> x+y;;
Partial Evaluation

- **Def:** To partially evaluate a (curried) function means to apply the function to some of its arguments but not to the rest
  - Example function: let add x y = x+y;;
  - Partially evaluated: (add 3)
  - Fully evaluated: (add 3 4)

- Partially evaluating a function of type $\tau_1 \to \tau_2 \to \ldots \to \tau_n$ always yields a new function of type $\tau_i \to \tau_{i+1} \to \ldots \to \tau_n$ (for some $i \in 2..n$)