

# CS 6371: Advanced Programming Languages

Dr. Kevin Hamlen

Fall 2008

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

# Patterns vs. Expressions

	Patterns	Expressions
	<b>Never evaluated</b>	<b>Evaluated</b>
x	a pattern that matches everything, introducing a new variable x	an expression that evaluates to the value of existing variable x
–	a pattern that matches everything	not a valid expression
x::t	a pattern that matches any list with at least one element, introducing new variables x and t	an expression that creates a new list by cons'ing element x onto list t
3	a pattern that matches the integer value 3	an expression that returns the integer value 3
3+4	not a valid pattern	evaluates to 7

# 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 equal

# Currying

- **Def:** A function is curried if none of its arguments has a tuple type.
  - Curried functions have types of the form  $\tau_1 \rightarrow \tau_2 \rightarrow \dots \rightarrow \tau_n$
  - The arrow type operator is right-associative, so whenever we write the above, it means  $\tau_1 \rightarrow (\tau_2 \rightarrow (\dots \rightarrow \tau_n))$
  - Function application is left-associative, so `(func a1 a2 ... an)` is short for `((func a1) a2) ... 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  $\tau_1 \rightarrow \tau_2 \rightarrow \dots \rightarrow \tau_n$  always yields a new function of type  $\tau_i \rightarrow \tau_{i+1} \rightarrow \dots \rightarrow \tau_n$  (for some  $i \in 2..n$ )