

CS 6371: Advanced Programming Languages

Dr. Kevin Hamlen

Fall 2007

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

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 $i \in 2..n$)

Modules & Separate Compilation

- Each OCaml module “Foo” consists of two files:
 - source file: foo.ml (like foo.c in C)
 - interface file: foo.mli (like foo.h in C)
- Interface files declare all public types and variables (including function variables):

```
type 'a btree = BNull | BNode ('a * 'a btree * 'a btree)
val tree2list : 'a btree -> 'a list
```

- Matching .ml and .mli files must have the SAME root filename (e.g. foo.ml goes with foo.mli)
- We will not be writing .mli files in this class, but you will need to be able to read the ones I provide you

Accessing Module Members

- To refer to a type, type constructor, or variable from an external module named “foo.ml”, use:

```
Foo.tree2list Foo.BNull;;  
Foo.tree2list (Foo.BNode (3, Foo.BNull, Foo.BNull));;
```

- Note: Module names MUST be capitalized!
- Alternatively, use “open” at the top of your .ml file to import all identifiers from the external module into the current namespace:

```
open Foo  
  
tree2list BNull;;  
tree2list BNode (3, BNull, BNull);;
```

Separate Compilation

- Compile each module separately with:

```
ocamlc -c foo.mli  
ocamlc -c foo.ml
```

- There must exist an .mli file for each module that foo.mli and foo.ml refer to
- These commands produce foo.cmi and foo.cmo, respectively
- Link all modules together with:

```
ocamlc -o output.exe foo.cmo bar.cmo ...
```

- Omit “.exe” when compiling on Unix
- Order of arguments matters! Each .cmo file may only depend on those that appear earlier in the command line

Auto-generating Interface Files

- If you type `ocamlc -c foo.ml` and there is no `foo.mli` file in the current directory, OCaml generates both a `foo.cmi` and a `foo.cmo`
- Auto-generated `.cmi` files export all top-level types and variables from your `.ml` file