

## CMSC 330, Spring 2009, Midterm 2 Practice Problems

### 1. Programming languages

- a. Describe the difference between OCaml modules and Java classes.

**Both provide a public definition for a group of functions whose internal details are hidden, but Java classes can also instantiate objects and inherit attributes from other classes (not possible with OCaml modules).**

- b. Describe the difference between strong and weak typing.

**Strong typing prevents types from being used interchangeably, weak typing allows types to be treated as other types through many implicit type conversions.**

- c. Explain how call-by-name simplifies implementing lazy evaluation.

**Expressions to be evaluated lazily may be passed as arguments to functions, since function arguments are not evaluated until used.**

- d. Describe the difference between an L-value and an R-value.

**L-values refer to the address of a symbol, R-values refer to the value for a symbol.**

- e. What is an activation record, and why is it usually allocated on a stack?

**An activation record contains state information for a function invocation. It is usually allocated on a stack so it can be easily freed upon function return by popping the stack.**

- f. Describe short circuiting.

**Short circuiting refers to programming language semantics that halts evaluation of the 2<sup>nd</sup> operand of a logical operator if the overall boolean result is already determined.**

### 2. Function arguments

For each code, explain whether  $g$  is an upward or downward funarg.

- a. `let f x = let g y = x + y in let app a b = a b in app g 1 ;;`

**$g$  is a downwards funarg since it is a function parameter passed to `app`**

- b. `let f x = let g y = x + y in g ;;`

**$g$  is an upwards funarg since it is a function return value for the 2<sup>nd</sup> `let`**

**A funarg is simply a function argument where the function is either**

**1. Passed as a parameter to a function call**

**2. Returned as the return value of a function call**

### 3. Static vs. Dynamic Scoping

Consider the following OCaml code.

```
let a = 1 ;;  
let f = fun () -> a ;; // value of a determined here for static scoping  
let a = 2 ;;  
f ();; // value of a determined here for dynamic scoping
```

- a. What value is returned by the invocation of `f()` with static scoping? Explain.  
**1, since the binding for “a” in the function “f = fun () -> a” refers to the closest lexical value of “a” at the point where the function is declared in the code (1<sup>st</sup> let a).**
- b. What value is returned by the invocation of `f()` with dynamic scoping? Explain.  
**2, since the binding for “a” in the function “f = fun () -> a” refers to the closest value of “a” in the call stack at the point where the function is actually invoked (2<sup>nd</sup> let a).**

Consider the following OCaml code.

```
let app f w = let x = 1 in f w ;; // value of x determined here  
// for dynamic scoping  
let add x y = let incr z = z+x in app incr y;; // value of x determined here  
// for static scoping  
(add 2 3) ;;
```

- c. What is the order of invocation for the functions `app`, `add`, and `incr` when evaluating the expression `(add 2 3)`?  
**1) add, 2) app, 3) incr**  
**incr is defined in add but not invoked until reaching the body of app (as f).**
- d. What value is returned by `(add 2 3)` with static scoping? Explain.  
**5, since the binding for x in the function *incr* refers to the closest lexical value of x (add x) at the point where the function is declared in the code.**
- e. What value is returned by `(add 2 3)` with dynamic scoping? Explain.  
**4, since the binding for x in the function *incr* refers to the closest value of x in the call stack (let x = 1) at the point where the function is actually invoked (by app f w ... in f w).**

#### 4. Parameter passing

Consider the following C code.

```
int i = 2;
void foo(int f, int g) {
    f = f-i;
    g = f;
}
int main() {
    int a[] = {2, 0, 1};
    foo(i, a[i]);
    printf("%d %d %d %d\n", i, a[0], a[1], a[2]);
}
```

- a. Give the output if C uses call-by-value

**2 2 0 1, since the call to foo() creates 2 local variables f & g (initialized with the values of i & a[i]), and all changes to f & g do not affect i or a[i].**

- b. Give the output if C uses call-by-reference

**0 2 0 0, since the call to foo() binds f to i & g to a[2], invoking foo() =**

```
void foo(f → i, g → a[2]) {
    f = f - i;           // equivalent to i = i - i → i = 0
    g = f;               // equivalent to a[2] = i → a[2] = 0
}
```

- c. Give the output if C uses call-by-name

**0 0 0 1, since the call to foo() replaces f with i & g with a[i], foo() =**

```
void foo(f → i, g → a[i]) {
    f = f - i;           // equivalent to i = i - i → i = 0
    g = f;               // equivalent to a[i] = i → a[0] = 0
}
```

#### 5. Lazy evaluation

Given the following OCaml code.

```
let doIf p x = if p then x else 0 ;;
let rec loop n = loop n ;;
doIf false (loop 0) ;;
```

- a. What is the result of evaluating the doIf expression if OCaml uses call-by-value?

**Infinite loop trying to evaluate loop 0 before its value is passed to doIf.**

- b. What is the result of evaluating the doIf expression if OCaml uses call-by-name?

**0, since loop 0 is directly passed to doIf and is not evaluated if p is false.**

- c. Rewrite the code (using thunks) so that the result of evaluating the doIf expression is the same as if OCaml used call-by-name, even though OCaml uses call-by-value.

```
let doIf p x = if (p ()) then (x ()) else 0
let rec loop n = loop n
doIf (fun () -> false) (fun () -> (loop 0))
```

6. Tail recursion

For each OCaml function, explain why it is or is not tail recursive

a. let rec foo x = 1 + (foo x)

**It is not tail recursive since 1 must be added to the value of foo x before it can be used as the return value of foo.**

b. let rec sum l = match l with

  [] -> 0

  l (x::xs) -> x + (sum xs)

**It is not tail recursive since x must be added to the return value of sum xs before it can be used as the return value of sum.**

c. let rec last = function

  [x] -> x

  l (\_::xs) -> last xs

**It is tail recursive since the value of the recursive call to “last xs” is also the return value.**

d. let rec fib x =

  if (x = 0) then 0

  else if (x = 1) then 1

  else (fib (x-1) + fib (x-2))

**It is not tail recursive since there are multiple calls to fib and none are the return value.**