

# CMSC330 Spring 2008 Homework #1 Solutions

1. (21 pts) OCaml Polymorphic Types

Consider a OCaml module Bst that implements a binary search tree:

```
module Bst = struct
  type bst =
    | Empty
    | Node of int * bst * bst

  let empty = Empty          (* empty binary search tree      *)

  let is_empty = function   (* return true for empty bst  *)
    | Empty -> true
    | Node (_, _, _) -> false

  let rec insert n = function (* insert n into binary search tree *)
    | Empty -> Node (n, Empty, Empty)
    | Node (m, left, right) ->
      if m = n then Node (m, left, right)
      else if n < m then Node(m, (insert n left), right)
      else Node(m, left, (insert n right))

  (* Implement the following functions
     val min : bst -> int
     val remove : int -> bst -> bst
     val fold : ('a -> int -> 'a) -> 'a -> bst -> 'a
     val size : bst -> int
     *)

  let rec min =              (* return smallest value in bst *)
  let rec remove n t =      (* tree with n removed          *)
  let rec fold f a t =      (* apply f to nodes of t in inorder *)
  let size t =              (* # of non-empty nodes in t    *)
end
```

a. (3 pts) Is insert tail recursive? Explain why or why not.

**No, since the return value for recursive call to insert cannot be used as the return value of the original call to insert. The return value is used to create a Node data type first, and the Node value is returned.**

b. (3 pts) Implement min as a tail-recursive function. Raise an exception for an empty bst. Any reasonable exception is fine.

```
let rec min = function
  | Empty -> (raise (Failure "min"))
  | Node (m, left, right) ->
    if (is_empty left) then m
    else min left
```

- c. (6 pts) Implement remove. The result should still be a binary search tree.

```

let rec remove n = function
  Empty -> Empty
  | Node (m, left, right) ->
    if m = n then (
      if (is_empty left) then right
      else if (is_empty right) then left
      else let x = min right in
        Node(x, left, remove x right)
      // OR
      // else let x = max left in
      // Node(x, remove x left, right)
    )
    else if n < m then Node(m, (remove n left), right)
    else Node(m, left, (remove n right))

```

- d. (6 pts) Implement fold as an inorder traversal of the tree so that the code

```
List.rev (fold (fun a m -> m::a) [] t)
```

will produce an (ordered) list of values in the binary search tree.

```

let rec fold f a n = match n with
  Empty -> a
  | Node (m, left, right) -> fold f (f (fold f a left) m) right

```

- e. (3 pts) Implement size using fold.

```
let size t = fold (fun a m -> a+1) 0 t
```

## 2. (36 pts) Recursive Descent Parser in OCaml

The example OCaml recursive descent parser 15-parseArith\_fact.ml employs a number of shortcuts. For instance, the function parseS handles the grammar rules for

$$S \rightarrow T + S \mid T$$

directly instead of first applying left factoring:

$$S \rightarrow T A \quad A \rightarrow + S \mid \text{epsilon}$$

However, we can still identify where code corresponding to parseA was inserted directly in the code for parseS, in the comments below:

```

let rec parseS lr =
  let x = parseT lr in
  match !lr with
  | ('+':t) ->
    lr := t;
    Sum (x, parseS lr)
  | _ -> x

```

(\* parseS \*)  
 (\* S → T A \*)  
 (\* parseA \*)  
 (\* if lookahead = First( + S ) \*)  
 (\* A → + S \*)  
 (\* A → epsilon \*)

Similarly, the function parseF handles the grammar rules for

$$F \rightarrow F ! \mid U$$

directly instead of rewriting the grammar, creating the following productions:

$$F \rightarrow ? \quad B \rightarrow ?$$

You must identify where code corresponding to parseB was inserted directly in the code for parseF in the comments below:

```
let rec parseF lr =
```

(\* parseF \*)

```

let rec fHelper lr tmp =
  match !lr with
  | (!'::t) -> (* parseB *)
    lr := t; (* 1: if lookahead = First( ? ) *)
    Fact (fHelper lr tmp) (* 2: ? → ? *)
  | _ -> tmp (* 3: ? → ? *)
in let x = parseU lr in (fHelper lr x) (* 4: ? → ? *)

```

- a. (3 pts) What rule should have been applied to the productions for F?

**Eliminate left recursion**

(e.g., change  $A \rightarrow A B \mid C$  to  $A \rightarrow C N$   
 $N \rightarrow B N \mid \text{epsilon}$ )

- b. (6 pts) What productions for F & B would be created by applying the rule?

**F → U B**

**B → ! B | epsilon**

- c. (3 pts) What sentential form should appear in place of ? in comment 1?

**! B**

- d. (3 pts) What production should appear in place of ? in comment 2?

**B → ! B**

- e. (3 pts) What production should appear in place of ? in comment 3?

**B → epsilon**

- f. (3 pts) What production should appear in place of ? in comment 4?

**F → U B**

3. (6 pts) Function arguments

- a. In the following code, identify each funarg and whether it is upward or downward.

```
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. In the following code, identify each funarg and whether it is upward or downward.

```
let f x = let g y = x + y in g ;;
```

**g is an upwards funarg since it is a function return value for 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**

**It is arguable that f can also be considered an upwards funarg since it is a function value bound to the symbol f by the 1<sup>st</sup> let, and accessible outside the scope of the let (since the binding is at the top level environment due to the “;;”).**

**In comparison, app is not considered a funarg since it is not used as a parameter or return value, nor is it accessible outside the scope of its let statement.**

**For the purposes of the homework (and the final), we’ll only consider funargs that are explicitly used as a function parameter or return value (i.e., g, not f).**

4. (6 pts) 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
```

- 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).**
- 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).**

5. (8 pts) 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]);
}
```

- (2 pts) 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].**
- (3 pts) 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
}
```
- (3 pts) 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
}
```

6. (10 pts) Polymorphism

Consider the following Java classes:

```

class A { public void a() { ... } }
class B extends A { public void b() { ... } }
class C extends B { public void c() { ... } }

```

Explain why the following code is or is not legal

- a. `int count(Set<A> s) { ... } ... count(new TreeSet<A>());`  
**Legal. Actual parameter type (Set<A>) matches formal parameter type (Set<A>)**
- b. `int count(Set<A> s) { ... } ... count(new TreeSet<B>());`  
**Illegal. Actual parameter type (Set<B>) is not a subclass of formal parameter type (Set<A>), even though B is a subclass of A.**
- c. `int count(Set s) { ... } ... count(new TreeSet<A>());`  
**Legal. Type erasure will cause formal parameter type (TreeSet<A>) to become TreeSet, which matches actual parameter type (Set).**
- d. `int count(Set<?> s) { ... } ... count(new TreeSet<A>());`  
**Legal. Actual parameter type (Set<A>) matches formal parameter type (Set<?>), since ? matches A.**
- e. `int count(Set<? extends A> s) { ... } ... count(new TreeSet<B>());`  
**Legal. Actual parameter type (Set<B>) matches formal parameter type (Set<? extends A>), since “? extends A” can match A and its subclasses B & C (classes that extend A, including A)**
- f. `int count(Set<? extends B> s) { ... } ... count(new TreeSet<A>());`  
**Illegal. Actual parameter type (Set<A>) does not match formal parameter type (Set<? extends B>), since “? extends B” can match only B and its subclass C (classes that extend B, including B)**
- g. `int count(Set<? extends B> s) { for (A x : s) x.a(); ... }`  
**Legal. The actual parameter type (Set<? extends B>) indicates s contains elements of class B or its subclasses. So any element of s may be treated as an object of class B or its subclasses (e.g., C). The for loop treats elements of s as objects of class A, which is a superclass of B, and thus is legal (can use subclass in place of superclass).**
- h. `int count(Set<? extends B> s) { for (C x : s) x.c(); ... }`  
**Illegal. The actual parameter type (Set<? extends B>) indicates s contains elements of class B or its subclasses. So any element of s may be treated as an object of class B or its subclasses (e.g., C). The for loop treats elements of s as objects of class C, and is illegal since elements of s may be objects of class B (cannot use superclass in place of subclass).**
- i. `int count(Set<? super B> s) { for (A x : s) x.a(); ... }`  
**Illegal. The actual parameter type (Set<? super B>) indicates s contains elements of class B or its superclasses. So any element of s may be treated as an object of class B or its superclasses (e.g., A, Object). The for loop treats elements of s as objects of class A, and is illegal since elements of s may be objects of class Object (cannot use superclass in place of subclass).**
- j. `int count(Set<? super B> s) { for (C x : s) x.c(); ... }`  
**Illegal. The actual parameter type (Set<? super B>) indicates s contains elements of class B or its superclasses. So any element of s may be treated as an object of class B or its superclasses (e.g., A, Object). The for loop**

**treats elements of s as objects of class C, which is not included and thus illegal.**

7. (6 pts) Java multithreading

- a. Using Java Conditions, you must implement a synchronization construct called MyBarrier. A MyBarrier object is created with a certain value n. When a thread calls the method enter( ), it enters the barrier and blocks until a total of n threads have entered the barrier. When the n<sup>th</sup> threads enters the barrier, all the threads waiting at the barrier wake up and unblock, and the n<sup>th</sup> thread continues without blocking. When a thread calls the method reset( ), the barrier is reset so that it starts fresh in counting up to n (i.e., n more threads must enter the MyBarrier).

```
public class MyBarrier {
    int num;           // shared read-only data
    int current = 0;   // shared modifiable data
    Lock lock = new ReentrantLock();
    Condition ready = lock.newCondition();

    public MyBarrier (int n) {
        num = n;
    }

    public void enter() throws InterruptedException {
        lock.lock();           // prevent data race on current
        current++;            // incr # of threads at barrier

        if (current == num) { // enough threads at barrier
            ready.signalAll(); // wake up other threads
        }                     // continue execution
        else {
            while (current < num) { // wait for more threads to enter
                ready.await();      // sleep until enough threads enter
            }                       // use while ( ) in case reset( ) called
        }
        lock.unlock();
    }

    public void reset() {
        lock.lock();           // prevent data race on current
        current = 0;
        lock.unlock();
    }
}
```

8. (6 pts) Garbage collection

Consider the following Java code.

```
Object a, b, c;
public foo() {
    a = new Object();    // object 1
    b = new Object();    // object 2
    c = new Object();    // object 3
    a = b;
    b = c;
    c = a;
}
```

- a. (3 pts) What object(s) are garbage when foo() returns? Explain why.

**Object 1 is garbage there are no longer any references to it within the program. After foo() returns, a → object 2, b → object 3, c → object 2.**

- b. (3 pts) Describe the difference between mark-and-sweep & stop-and-copy.

**Mark-and-sweep stops the program to determine what objects are still reachable. Stop-and-copy in addition will move reachable objects to new locations.**

9. (4 pts) Markup languages

- a. Creating your own XML tags, write an XML document that organizes the following information: 1-hour test on Spanish Monday in Jiménez worth 15%. 1-hour test on Computers Tuesday in CSIC worth 10%. 30-minute test on Computers Friday in AVW worth 5%.

```
<testList>
  <test>
    <length>1 hour</length>
    <subject>Spanish</subject>
    <date>Monday</date>
    <location>Jiménez</location>
    <value>15% </value>
  </test>
  <test>
    <length>1 hour</length>
    <subject>Computers</subject>
    <date>Tuesday</date>
    <location>CSIC</location>
    <value>10% </value>
  </test>
  <test>
    <length>30 minute</length>
    <subject>Computers</subject>
    <date>Friday</date>
    <location>AVW</location>
    <value>5% </value>
  </test>
</testList>
```

```

</test>
</testList>
                                or
<testList>
  <test subject="Spanish">
    <length unit="hour">1</length>
    ...
  </test>
  <test subject="Computers" >
    <length unit="hour">1</length>
    ...
  </test>
  <test subject="Computers" >
    <length unit="minute">30</length>
    ...
  </test>
</testList>

```

10. (8 pts) Lambda calculus

Evaluate the following  $\lambda$ -expressions as much as possible

- a.  $(\lambda z.z) (\lambda y.y y) (\lambda x.x a) \rightarrow$  //  **$\beta$ -reduction = body[sym/replacement]**  
 $(\lambda z.z) (\lambda y.y y) (\lambda x.x a) \rightarrow$  //  $z[z/(\lambda y.y y)]$  replace  $z$  with  $\lambda y.y y$   
 $(\lambda y.y y) (\lambda x.x a) \rightarrow$  //  $y y[y/(\lambda x.x a)]$  replace  $y$  with  $\lambda x.x a$   
 $(\lambda x.x a) (\lambda x.x a) \rightarrow$  //  $x a[x/(\lambda x.x a)]$  replace  $x$  with  $\lambda x.x a$   
 $(\lambda x.x a) a \rightarrow a a$  //  $x a[x/a]$  replace  $x$  with  $a$
- b.  $(\lambda z.z) (\lambda z.z z) (\lambda z.z y) \rightarrow$  //  **$\beta$ -reduction: replace  $z$  with  $\lambda z.z z$**   
 $(\lambda z.z z) (\lambda z.z y) \rightarrow$  //  **$\beta$ -reduction: replace  $z$  with  $\lambda z.z y$**   
 $(\lambda z.z y) (\lambda z.z y) \rightarrow$  //  **$\beta$ -reduction: replace  $z$  with  $\lambda z.z y$**   
 $(\lambda z.z y) y \rightarrow y y$  //  **$\beta$ -reduction: replace  $z$  with  $y$**
- c.  $(\lambda x.\lambda y.x y y) (\lambda a.a) b \rightarrow$  //  **$\beta$ -reduction: replace  $x$  with  $\lambda a.a$**   
 $(\lambda x.\lambda y.x y y) (\lambda a.a) b \rightarrow$  //  **$\beta$ -reduction: replace  $y$  with  $b$**   
 $(\lambda a.a) b b \rightarrow b b$  //  **$\beta$ -reduction: replace  $a$  with  $b$**
- d.  $(\lambda x.\lambda y.x y y) (\lambda y.y) y \rightarrow$  //  **$\alpha$ -conversion: replace  $y$  with  $a$**   
 $(\lambda x.\lambda a.x a a) (\lambda y.y) y \rightarrow$  //  **$\beta$ -reduction: replacing  $x$  with  $\lambda y.y$**   
 $(\lambda a.(\lambda y.y) a a) y \rightarrow$  //  **$\beta$ -reduction: replacing  $a$  with  $y$**   
 $(\lambda y.y) y y \rightarrow y y$  //  **$\beta$ -reduction: replacing  $y$  with  $y$**

11. (24 pts) Lambda calculus

Prove the following using the appropriate  $\lambda$ -calculus encodings

a.  $\text{not} (\text{not true}) = \text{true}$

**Given:**

$\text{not} = \lambda x.((x \text{ false}) \text{ true})$

$\text{true} = \lambda x.\lambda y.x$

$\text{false} = \lambda x.\lambda y.y$

**Proof:**

$\text{not} (\text{not true})$

$= \lambda x.((x \text{ false}) \text{ true}) (\text{not true})$

$= ((\text{not true}) \text{ false}) \text{ true}$

$= ((\lambda x.((x \text{ false}) \text{ true}) \text{ true}) \text{ false}) \text{ true}$

$= (((\text{true} \text{ false}) \text{ true}) \text{ false}) \text{ true}$

$= (((\lambda x.\lambda y.x) \text{ false}) \text{ true}) \text{ false}) \text{ true}$

$= (((\lambda y.\text{false}) \text{ true}) \text{ false}) \text{ true}$

$= ((\text{false}) \text{ false}) \text{ true}$

$= ((\lambda x.\lambda y.y) \text{ false}) \text{ true}$

$= (\lambda y.y) \text{ true}$

$= \text{true}$

*// replacing 1<sup>st</sup> not w/ encoding*

*//  $\beta$ -reduction:  $x \rightarrow \text{not true}$*

*// replacing not w/ encoding*

*//  $\beta$ -reduction:  $x \rightarrow \text{true}$*

*// replace true w/ encoding*

*//  $\beta$ -reduction: 1<sup>st</sup>  $x \rightarrow \text{false}$*

*//  $\beta$ -reduction:  $y \rightarrow \text{true}$*

*// replace false w/ encoding*

*//  $\beta$ -reduction:  $x \rightarrow \text{false}$*

*//  $\beta$ -reduction:  $y \rightarrow \text{true}$*

*// not (not true) = true*

b.  $\text{if false then } x \text{ else } y = y$

**Given:**

$\text{if } a \text{ then } b \text{ else } c = a b c$

$\text{true} = \lambda x.\lambda y.x$

$\text{false} = \lambda x.\lambda y.y$

**Proof:**

$\text{if false then } x \text{ else } y$

$= \text{false } x y$

$= (\lambda x.\lambda y.y) x y$

$= (\lambda y.y) y$

$= y$

*// replacing if... w/ encoding*

*// replacing false w/ encoding*

*//  $\beta$ -reduction:  $x \rightarrow x$*

*//  $\beta$ -reduction:  $y \rightarrow y$*

*// if false then x else y = y*

c.  $\text{succ } 2 = 3$

**Given:**

$2 = \lambda f.\lambda y.f (f y)$

$3 = \lambda f.\lambda y.f (f (f y))$

$\text{succ} = \lambda z.\lambda f.\lambda y.f (z f y)$

**Proof:**

$\text{succ } 2$

$= (\lambda z.\lambda f.\lambda y.f (z f y)) 2$

$= \lambda f.\lambda y.f (2 f y)$

$= \lambda f.\lambda y.f ((\lambda f.\lambda y.f (f y)) f y)$

$= \lambda f.\lambda y.f ((\lambda y.f (f y)) y)$

$= \lambda f.\lambda y.f (f (f y))$

$= 3$

*// replacing succ w/ encoding*

*//  $\beta$ -reduction:  $z \rightarrow 2$*

*// expanding 2 w/ encoding*

*//  $\beta$ -reduction: 1<sup>st</sup>  $f \rightarrow f$*

*//  $\beta$ -reduction: 1<sup>st</sup>  $y \rightarrow y$*

*// apply encoding for 3*

*// succ 2 = 3*

d.  $(* 1 3) = 3$

**Given:**

$$\begin{aligned} 1 &= \lambda f. \lambda y. f y \\ 3 &= \lambda f. \lambda y. f (f (f y)) \\ M * N &= \lambda x. (M (N x)) \end{aligned}$$

**Proof:**

$$\begin{aligned} (* 1 3) & && // \text{replacing } * \text{ w/ encoding} \\ = \lambda x. (1 (3 x)) & && // \text{replacing } 3 \text{ w/ encoding} \\ = \lambda x. (1 (\lambda f. \lambda y. f (f (f y)) x)) & && // \beta\text{-reduction: } 1^{\text{st}} f \rightarrow x \\ = \lambda x. (1 (\lambda y. x (x (x y)))) & && // \text{replacing } 1 \text{ w/ encoding} \\ = \lambda x. ((\lambda f. \lambda y. f y) (\lambda y. x (x (x y)))) & // \beta\text{-reduction: } 1^{\text{st}} f \text{ w/ } \lambda y. x (x (x y)) \\ = \lambda x. (\lambda y. (\lambda y. x (x (x y)))) y & // \beta\text{-reduction: } 1^{\text{st}} y \rightarrow y \\ = \lambda x. \lambda y. x (x (x y)) & // \alpha\text{-conversion: replace } x \text{ with } f \\ = \lambda f. \lambda y. f (f (f y)) & // \text{apply encoding for } 3 \\ = 3 & \end{aligned}$$

e.  $(+ 2 1) = 3$

**Given:**

$$\begin{aligned} 1 &= \lambda f. \lambda y. f y \\ 2 &= \lambda f. \lambda y. f (f y) \\ 3 &= \lambda f. \lambda y. f (f (f y)) \\ M + N &= \lambda x. \lambda y. (M x)((N x) y) \end{aligned}$$

**Proof:**

$$\begin{aligned} (+ 2 1) & && // \text{replacing } + \text{ w/ encoding} \\ = \lambda x. \lambda y. (2 x)((1 x) y) & && // \text{replacing } 2 \text{ w/ encoding} \\ = \lambda x. \lambda y. ((\lambda f. \lambda y. f (f y)) x)((1 x) y) & && // \beta\text{-reduction: } 1^{\text{st}} f \rightarrow x \\ = \lambda x. \lambda y. (\lambda y. x (x y))((1 x) y) & && // \text{replacing } 1 \text{ w/ encoding} \\ = \lambda x. \lambda y. (\lambda y. x (x y))(((\lambda f. \lambda y. f y) x) y) & // \beta\text{-reduction: } 1^{\text{st}} f \rightarrow x \\ = \lambda x. \lambda y. (\lambda y. x (x y))((\lambda y. x y) y) & // \beta\text{-reduction: } 3^{\text{rd}} y \rightarrow y \\ = \lambda x. \lambda y. (\lambda y. x (x y))(x y) & // \beta\text{-reduction: } 2^{\text{nd}} y \rightarrow x y \\ = \lambda x. \lambda y. x (x (x y)) & // \alpha\text{-conversion: replace } x \text{ with } f \\ = \lambda f. \lambda y. f (f (f y)) & // \text{apply encoding for } 3 \\ = 3 & \end{aligned}$$

f.  $(Y \text{ fact}) 2 = 2$  // you do not need to expand any operators except fact & Y

**Given:**

$$\begin{aligned} Y &= \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x)) \\ \text{fact} &= \lambda f. \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * (f (n-1)) \end{aligned}$$

**Proof:**

$$\begin{aligned} (Y \text{ fact}) 2 & && // \text{replacing } Y \text{ w/ encoding} \\ = (\lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))) \text{ fact} 2 & && // \beta\text{-reduction: } 1^{\text{st}} f \rightarrow \text{fact} \\ = (\lambda x. \text{fact} (x x)) (\lambda x. \text{fact} (x x)) 2 & // \beta\text{-reduction: } 1^{\text{st}} x \rightarrow \lambda x. \text{fact} (x x) \\ = (\text{fact} ((\lambda x. \text{fact} (x x)) (\lambda x. \text{fact} (x x)))) 2 & \\ & // \text{apply encoding for } (Y \text{ fact}) \\ & // ((\lambda x. \text{fact} (x x)) (\lambda x. \text{fact} (x x))) \rightarrow (Y \text{ fact}) \\ & // \text{we know this is the encoding for } (Y \text{ fact}) \text{ from } 3^{\text{rd}} \text{ line of proof} \\ = (\text{fact} (Y \text{ fact})) 2 & // \text{apply encoding for fact} \end{aligned}$$

```

= (λf. λn.if n = 0 then 1 else n * (f (n-1))) (Y fact) 2
// β-reduction: 1st f → (Y fact)
= (λn.if n = 0 then 1 else n * ((Y fact) (n-1))) 2 // β-reduction: n → 2
= if 2=0 then 1 else 2 * ((Y fact) (2-1)) // apply if
= 2 * ((Y fact) 1) // showed in class (Y fact) 1 = 1
= 2 * 1 // apply *
= 2

```

12. (27 pts) Miscellaneous

- 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. Describe the difference between ad-hoc and parametric polymorphism.  
**Ad hoc polymorphism applies to code supporting a finite range of types whose combinations must be specified, parametric polymorphism applies to code written without mention to type that can transparently support an arbitrary number of types.**
- f. Describe the difference between starvation and deadlock.  
**Deadlocked threads are halted waiting for each other's locks, whereas starving threads are waiting for locks from other (non-starving) threads.**
- g. Describe how functional programming may be used to simulate OOP.  
**An object may be simulated as a tuple, where each element of the tuple is a closures representing a method for the object.**
- h. Describe the difference between HTML and XML.  
**HTML tags are predefined and presentation-oriented, whereas XML tags are user defined and are intended for describing data and metadata.**