CS 603 Exam 2 **Spring 2007** **Solution**

1. Explain the differences between (i) imperative programming, (ii) object-oriented programming, and (iii) functional programming. Specify the relevant features that are characteristic of each paradigm, that is, the features that must be present in any programming language that fully supports the paradigm. **[9 points]**

   Imperative programming is characterized by the use of statements, such as assignment and iteration, which use side effects to change the program's state as stored in memory. Functional programming is characterized by defining and calling functions, without state or side effects, with heavy use of recursion, higher-order functions, and single-valued variables. Object-oriented programming is characterized by a collection of interacting objects, each of which maintains its own state, and which are instances of abstract data types that are defined using inheritance, polymorphism, and late method binding.

2. Describe how class variables and class methods are different from instance variables and instance methods in an object-oriented programming language such as Smalltalk, C++, or Java. **[8 points]**

   A class variable is allocated just once and shared by all the instances of the class, but a new copy of each instance variable is allocated every time a new instance of the class is created. A class method is invoked on the class rather than an instance, and it can only access class variables. An instance method is invoked on an instance (the receiver object), and it can also access instance variables of the receiver.

3. *Pass-by-need* is a kind of parameter passing such that evaluation of each actual parameter is delayed until the value of the formal parameter is needed. If and when this evaluation does occur, the result will be saved for any subsequent evaluations of the formal parameter, so that each actual parameter is evaluated at most once. Write an example in ML or μScheme such that the result would be different if using pass-by-need rather than pass-by-value, and state the result in each case. **[7 points]**

   First example:

ML	**μScheme**
fun f _ = 0; **val x = ref 1;** **f (x := 2);** **!x;**	**(define f (a) 0)** **(val x 1)** **(f (set x 2))** **x**

   The result from the last line is 1 using pass-by-need, or 2 with pass-by-value. [In some ways, pass-by-need is similar to passing a lambda expression or a block.]

Second example:

ML	**μScheme**
val x = ref 1;	**(val x 1)**
val y = ref 2;	**(val y 2)**
if !x < !y then x:= 3 else y:= 4;	**(if (< x y) (set x 3) (set y 4))**
!x;	**x**
!y;	**y**

The result of the last two lines are 3, 2 using pass-by-need, or 3, 4 with pass-by-value. [ML and Scheme do use pass-by-need for some predefined functions such as if.]

4. Write a suffixes function in ML that works as follows:
suffixes [10, 20, 30, 40] returns [[10, 20, 30, 40], [20, 30, 40], [30, 40], [40], [ ]].

   a. First write suffixes using recursion and pattern matching. Do not use map, foldr, or foldl. **[6 points]**

```
fun suffixes [ ] = [[ ]]
  |   suffixes (h::t) = (h::t) :: suffixes t;
```

   b. Next write suffixes using map, foldr, and/or foldl. Do not use any explicit recursion. **[6 points]**

```
fun suffixes L = foldr (fn (x,y) => (x::hd y)::y) [[ ]] L;
```

   c. Specify the polymorphic type of the suffixes function. **[3 points]**

```
fn : 'a list -> 'a list list
```

5. Write a filter function in ML that works as follows:
filter (fn x => x>0, [3, ~2, 5, 0, 6, ~4, 7]) returns [3, 5, 6, 7].

   a. First write filter using recursion and pattern matching. Do not use map, foldr, or foldl. **[6 points]**

```
fun filter (f, [ ]) = [ ]
  |   filter (f, h::t) = if f h then h::filter(f, t) else filter(f, t);
```

   b. Next write filter using map, foldr, and/or foldl. Do not use any explicit recursion. **[6 points]**

```
fun filter (f, L) = foldr op@ [ ] (map (fn x => if f x then [x] else [ ]) L);
```

   c. Specify the polymorphic type of the filter function. **[3 points]**

```
fn : ('a -> bool) * 'a list -> 'a list
```

6. Write a superExp function in ML, such that superExp(x, k) computes x^{2^k}.
Example: superExp(3.0, 4) returns $3.0^{2^4} = 3.0^{16} = 43046721.0$. Assume that $k \geq 0$. For full credit, make only one recursive call within your function, and do not use any helper functions. Also, your function should be efficient, that is, superExp(x, k) should perform a total of only k multiplication steps.
Hints: think of x^{2^k} as $(\ldots(x^2)^2\ldots)^2$, and use a let expression.

   a. Write the superExp function. **[9 points]**

```
fun superExp(x, 0) = x:real
  |   superExp(x, k) = let val y = superExp(x, k-1) in  y*y end;
```

   b. Specify the type of the superExp function. **[2 points]**

fn : real * int -> real

7. Write a compare function in ML that works as follows:
compare op<= [1,2,5] [1,3,4] returns [Equal, Less, Greater].
compare (fn ((a,b),(c,d)) => a<=c andalso b<=d)
[(1,2), (3,5), (4,6), (7,8)] [(1,2), (4,6), (3,5), (8,7)]
returns [Equal, Less, Greater, Incompatible].

   a. First write a datatype that is needed by the compare function. **[3 points]**

```
datatype Order = Equal | Less | Greater | Incompatible;
```

   b. Next write the compare function. **[12 points]**

```
fun compare _ [ ] [ ] = [ ]
  |  compare le (w::x) (y::z) =
        if le(w,y) andalso le(y,w) then Equal::compare le x z
        else if le(w,y) then Less::compare le x z
        else if le(y,w) then Greater::compare le x z
        else Incompatible::compare le x z;
```

   c. Specify the polymorphic type of the compare function. **[5 points]**

fn : ('a * 'a -> bool) -> 'a list -> 'a list -> Order list

8. Write a µSmalltalk class called Interval such that each instance represents a continuous finite range on the real number line. Here is a client of class Interval which illustrates how the class Interval should operate. **[25 points]**

```
(val X (left:right: Interval 0 4))      ; X is the interval 0 … 4
(val Y (left:right: Interval 2 7))      ; Y is the interval 2 … 7
(val Z (left:right: Interval 5 11))     ; Z is the interval 5 … 11
(length X)                              ; returns 4
(length Y)                              ; 5
(length Z)                              ; 6
(disjoint: X Y)                         ; <False>
(disjoint: X Z)                         ; <True>
(disjoint: Y X)                         ; <False>
(disjoint: Y Z)                         ; <False>
(disjoint: Z X)                         ; <True>
(disjoint: Z Y)                         ; <False>
(intersects: X Y)                       ; <True>
(intersects: X Z)                       ; <False>
(intersects: Y X)                       ; <True>
(intersects: Y Z)                       ; <True>
(intersects: Z X)                       ; <False>
(intersects: Z Y)                       ; <True>
```

```
(class Interval Object (left right)
        (classMethod left:right: (L R)
                (locals i)
                (set i (new self))
                (left: i L)
                (right: i R)
                i)
        (method length ( )
                (- right left))
        (method disjoint: (other)
                (| (> left (right other))
                   (< right (left other))))
        (method intersects: (other)
                (& (<= left (right other))
                   (>= right (left other))))
        (method left ( ) left)
        (method right ( ) right)
        (method left: (L) (set left L))
        (method right: (R) (set right R))
)
```